Technology
Arts Sciences
TH Köln

CloudRun

Analyse und Erweiterung

Bachelorarbeit

ausgearbeitet von

Andreas Pahlen

zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

vorgelegt an der

Technischen Hochschule Köln
Campus Gummersbach
Fakultät für Informatik und
Ingenieurwissenschaften

im Studiengang
Informatik

Erster Prüfer: Prof. Dr. Martin Eisemann
Technische Hochschule Köln

Zweiter Prüfer: Prof. Dr. Christian Kohls
Technische Hochschule Köln

Gummersbach, im Mai 2017

Adressen: Andreas Pahlen
Feldstraße 9
51643 Gummersbach
mail@apahlen.de

Prof. Dr. Martin Eisemann
Technische Hochschule Köln
Institut für Informatik
Steinmüllerallee 1
51643 Gummersbach
martin.eisemann@th-koeln.de

Prof. Dr. Christian Kohls
Technische Hochschule Köln
Institut für Informatik
Steinmüllerallee 1
51643 Gummersbach
christian.kohls@th-koeln.de

Kurzfassung

Diese Bachelorarbeit „CloudRun - Analyse und Erweiterung“ von Andreas Pahlen behandelt die Analyse der Software „CloudRun“ [Pahlen u. Förster, 2017] sowie dessen Einbindung in eine Programmiersprache am Beispiel von Python.

Ziel der Analyse ist Herauszufinden, ob und in welchem Maße sich CloudRun für den produktiven Einsatz eignet, bzw. welche Änderungen nötig sind um einen sicheren und effizienten Einsatz zu ermöglichen.

Die Einbindung CloudRuns in die Programmiersprache Python dient als Beispiel und Referenz zur Erstellung weiterer Frameworks und ermöglicht zugleich den Einsatz eines CloudRun-Servers für produktive Zwecke.

Abstract

This bachelor thesis „CloudRun - Analysis and Extension“ by Andreas Pahlen deals with the analysis of the software „CloudRun“ [Pahlen u. Förster, 2017] as well as its integration into a programming language using the example of Python.

The aim of the analysis is to find out whether and to what extent CloudRun is suitable for productive use or what changes are necessary to ensure a safe and efficient usage.

The inclusion of CloudRun in the Python programming language serves as an example and reference for the creation of additional libraries and also allows the use of a CloudRun-server for productive purposes.

Inhaltsverzeichnis

1 Einleitung

CloudRun, eine Serveranwendung zur entfernten Ausführung von Algorithmen und Software, wurde im Rahmen des Praxisprojektes von Andreas Pahlen und Stefan Förster entwickelt [Pahlen u. Förster, 2017]. Zwar wurde während der Entwicklung CloudRuns auf Leistungsfähigkeit und Sicherheit geachtet, jedoch wurden diese Eigenschaften bisher nicht überprüft, daher ist eine Analyse vor der Verwendung CloudRuns im produktiven Umfeld unabdingbar. Zudem soll CloudRun vom Endanwender so einfach wie möglich verwendet werden können, weshalb auch die Notwendigkeit für eine benutzerfreundliche Anwendungsbibliothek besteht.

1.1 Was ist CloudRun?

Um die behandelten Themen in dieser Bachelorarbeit vollständig nachvollziehen zu können, muss zunächst bekannt sein, was CloudRun ist und wie der interne Programmaufbau aussieht.

CloudRun ist eine mit Hilfe von NodeJS [NodeJS] entwickelte Serveranwendung, welche eine Schnittstelle zum Aufrufen von sogenannten Modulen zur Verfügung stellt. Dies ermöglicht die Verwendung von Algorithmen oder Software über eine Netzwerk- oder Internetverbindung, ohne dass der Anwender selbst direkten Zugriff auf die bereitgestellten Module hat. Dadurch lassen sich beispielsweise Algorithmen zu Forschungszwecken bereitstellen, selbst wenn der Quelltext nicht öffentlich einsehbar sein soll. Zwar ließe sich dieses Verhalten auch durch das Verteilen von ausführbaren, bereits kompilierten Dateien realisieren, jedoch besteht dann die Gefahr des sogenannten Dekompilierens [Kumar, 2011], wodurch kompilierte Dateien relativ zuverlässig zurück in Quelltext übersetzt werden können. Dadurch, dass CloudRun jedoch als Vermittlung zwischen Anwender und Modul agiert, besteht kein direkter Zugriff auf den Quelltext oder ausführbare Datei des Algorithmus'.

Weiterhin bietet CloudRun den Vorteil, dass die Notwendigkeit der Beschaffung und des Importierens von Algorithmen in Anwenderprojekte entfällt. Dies minimiert den

Aufwand bei der Realisierung von Projekten, welche fremde Algorithmen verwenden sollen.

Da der Sinn und Zweck von CloudRun nun bekannt ist, wird im Folgenden in groben Zügen auf den internen Aufbau eingegangen. Für eine detaillierte Beschreibung aller Bestandteile CloudRuns, ziehen Sie bitte die Projektdokumentation [Pahlen u. Förster, 2017] hinzu.

Die Kommunikation mit dem CloudRun Server erfolgt über eine REST-Schnittstelle [Fielding, 2000], dabei werden die Daten im JSON-Format [Internet Engineering Task Force (IETF), 2014] übertragen.

CloudRun ist modular aufgebaut, sodass die Erweiterung der Funktionalität durch den Administrator oder andere Entwickler so einfach wie möglich ist. Es gibt insgesamt drei Arten von modularen Bestandteilen CloudRuns, welche im Folgenden beschrieben werden.

Module Die sogenannten Module beinhalten die Algorithmen, bzw. übernehmen den Aufruf einer ausführbaren Datei. Module nehmen die vom Anwender übermittelten Parameter zur Ausführung des Algorithmus' entgegen, berechnen das Ergebnis und geben es an den CloudRun Server zurück, welcher anschließend das Ergebnis zurück an den Anwender übermittelt.

Alle Module werden in einem Ordner (`modules`) abgelegt und automatisch beim Start des CloudRun Servers identifiziert und in den Funktionsumfang des Servers eingebunden.

Authentifizierungsmodule CloudRun bietet die Möglichkeit der sogenannten Tokenauthentifizierung. Dadurch ist es nur autorisierten Nutzern möglich (welche über einen Token verfügen) den CloudRun Server zu verwenden. Wird beim Aufruf eines Moduls ein Token übermittelt, welcher dem Server nicht bekannt ist, so wird die Anfrage mit einer Fehlermeldung quittiert, das Modul jedoch nicht ausgeführt.

Da es verschiedene Möglichkeiten zur Tokenverwaltung gibt, bietet CloudRun standardmäßig drei verschiedene Authentifizierungsmodule:

- `no_auth`: Keine Authentifizierung, jeder kann den Server verwenden

- `textfile_auth`: Die Tokens sind in einer Textdatei gespeichert, getrennt durch Zeilenumbrüche
- `mysql_auth`: Die Tokens sind in einer MySQL-Datenbank gespeichert

Sollte der Administrator des CloudRun Servers eine abweichende Speicherung der Tokens wünschen, so lässt sich dies durch die Eigenentwicklung eines Authentifizierungsmoduls realisieren.

Handler Handler dienen der Entgegennahme von Anfragen über die REST Schnittstelle. Der Standardhandler `request` nimmt beispielsweise Anfragen an Module entgegen, prüft die Authentifizierung, führt das Modul aus und übermittelt das Ergebnis zurück an den Anwender. Weiterhin verfügt CloudRun standardmäßig über den Handler `server_info`, welcher Informationen über die auf dem Server installierten Module liefert.

Um zu verdeutlichen, wie die verschiedenen Module miteinander zusammenspielen, folgt nun eine Beispielanfrage. Dabei wird das Modul „`stringAnalyzer`“ aufgerufen, welches CloudRun standardmäßig beinhaltet.

```
{
  "token":  "xyz",
  "module": "stringAnalyzer",
  "data":   "Dies ist ein Test!"
}
```

In diesem Beispiel lautet der Authentifizierungstoken „xyz“, welcher an das verwendete Authentifizierungsmodul übergeben wird. Das Authentifizierungsmodul prüft, ob der Token bekannt (also zugelassen) ist. Ist die Authentifikation erfolgreich, so werden die übermittelten Daten „Dies ist ein Test!“ an das Modul `stringAnalyzer` übergeben.

Diese Anfrage wird per POST-Request an die URL `http://cloudrun-server.de/request/` geschickt. Dabei wird deutlich, dass die aufzurufende URL den zu verwendenden Handler beinhaltet, in diesem Falle `request`.

Der CloudRun Server würde bei diesem Beispiel mit dem Ergebnis des Moduls `stringAnalyzer` antworten:

```
{
  "length": 18,
  "letter_count": {
    "D": 1,
    "i": 3,
    "e": 3,
    "s": 3,
    " ": 3,
    "t": 2,
    "n": 1,
    "T": 1,
    "!": 1
  }
}
```

CloudRun wurde nach der Entwicklung unter der Apache 2.0 Open Source Lizenz veröffentlicht und kann über das github-Repository frei heruntergeladen werden: `https://github.com/Organized92/cloudrun`

1.2 Motivation

Der potentielle Nutzen von CloudRun ist groß, daher wäre die Möglichkeit CloudRun produktiv einsetzen zu können eine Bereicherung, insbesondere für akademische Zwecke. Da der Einsatz einer ungeprüften Software, in Hinblick auf Zuverlässigkeit und Sicherheit, jedoch mit enormen Risiken verbunden ist, muss das Projekt zunächst gründlich analysiert werden. Eine solche Analyse schafft Klarheit darüber, ob ein CloudRun Server ruhigen Gewissens bereitgestellt werden kann, oder nicht.

Aus diesem Grund befasst sich diese Bachelorarbeit mit der Frage, ob CloudRun bereit für den produktiven Einsatz ist, bzw. welche Anpassungen getroffen werden müssen, damit dies der Fall ist.

Ein CloudRun Server ist ohne die entsprechende Anbindung an populäre Programmiersprachen jedoch nur mit großem Aufwand nutzbar, was die Sinnhaftigkeit eines solchen Servers wiederum enorm schmälert. Aus diesem Grund besteht die Notwendigkeit von Bibliotheken, welche die Kommunikation mit dem CloudRun Server übernehmen, um den Aufwand der Benutzung von CloudRun so gering wie möglich zu halten.

Daher behandelt diese Bachelorarbeit ebenso den theoretischen Aufbau solcher Bibliotheken und verdeutlicht dies an einer Implementierung für die Programmiersprache Python.

Die Kombination dieser beiden Teilaufgaben wird den produktiven Einsatz von CloudRun ermöglichen, was enorm zum Wert der Software beiträgt. Da CloudRun unter einer OpenSource-Lizenz bereitgestellt wird, erhält die Öffentlichkeit dadurch Zugang zu einer einsetzbaren Serveranwendung, welche insbesondere im akademischen Umfeld wertvolle Dienste leisten kann.

1.3 Vergleichbare Software / Related Work

Nach der Beendigung des Praxisprojekts, also der Entwicklung von CloudRun, wurde ich auf ein sehr ähnliches Projekt aufmerksam [Algorithmia]. Algorithmia bietet den Zugriff auf verschiedene Algorithmen über eine REST-Schnittstelle und verfolgt damit genau den gleichen Ansatz wie CloudRun.

CloudRun wurde ohne Kenntnisse über Algorithmia entwickelt, jedoch ist auffällig, dass auch Algorithmia auf die Übertragung der Daten im JSON-Format setzt. Dies spricht dafür, dass wir während der Entwicklung CloudRuns korrekte bzw. gängige Ansätze verfolgt haben. Ebenso erfolgt die Authentifizierung bei Algorithmia über einen API-Key, also einen Token.

Im Gegensatz zu CloudRun ist Algorithmia jedoch keine frei verfügbare Software. Vielmehr handelt es sich dabei um einen Dienstleister, welcher einen „offenen Marktplatz für Algorithmen“ (freie Übersetzung von „Open Marketplace for Algorithms“, der Selbstbeschreibung Algorithmias) sowie die dazugehörige Infrastruktur bereitstellt. „Offen“ bedeutet dabei, dass dort jeder Entwickler seine Algorithmen zur Verfügung stellen kann, entweder kostenlos oder gegen eine Gebühr pro Aufruf. Zudem wird eine Gebühr für die benötigte Rechenzeit erhoben.

Algorithmia bietet nach eigenen Angaben den Zugriff auf über 3000 „Microservices“. Dabei handelt es sich bei den Bekanntesten unter Anderem um eine Gesichtserkennung, eine Stimmungserkennung in Texten sowie eine Überprüfung von Bildern auf Nacktheit.

Erreichbar ist Algorithmia über 16 verschiedene Programmiersprachen, darunter Python, Java, JavaScript sowie .NET-Sprachen. Zudem wird auch die manuelle Kommunikation mittels `curl` beworben.

Vergleichen lassen sich Algorithmia und CloudRun in sofern, dass die technische Umsetzung grundsätzlich sehr ähnlich ist. Algorithmia basiert allerdings auf dem Ansatz, einen Marktplatz zu schaffen, auf dem Entwickler von Algorithmen – wenn sie das möchten – Geld verdienen können. Zudem verdient Algorithmia an jeder Sekunde Rechenzeit ebenfalls einen kleinen Betrag. Die von Algorithmia eingesetzte Software ist nicht frei verfügbar und kann daher, im Gegensatz zu CloudRun, nicht für persönliche Zwecke eingesetzt werden.

Für kleine Projekte und Hobbyentwickler bietet Algorithmia jedoch einen kostenlosen Tarif, welcher pro Monat 5.000 Sekunden Rechenzeit (bei ausschließlicher Verwendung kostenloser Algorithmen) bereitstellt.

2 Analyse

Während der Entwicklungsphase von CloudRun wurde zwar darauf geachtet, dass die Software möglichst leistungsfähig und sicher arbeitet, jedoch wurde CloudRun bisher nicht explizit bezüglich seiner Zuverlässigkeit beziehungsweise Sicherheit analysiert. Vor dem Einsatz einer Serversoftware sollte eine solche Analyse jedoch dringend durchgeführt werden, um die Verfügbarkeit und Integrität [Bundesamt für Sicherheit in der Informationstechnik] des Servers im Ganzen nicht zu gefährden.

Grundsätzlich lässt sich eine solche Analyse einer Serveranwendung in zwei grobe Bereiche aufteilen; die Zuverlässigkeits- und Sicherheitsanalyse. Die Zuverlässigkeitsanalyse erlaubt ein Urteil über die Belastbarkeit einer Software, nur bedingt abhängig von der Rechenleistung des ausführenden Computers. Dadurch kann ermittelt werden, ob die Software durch übermäßige Belastung abstürzt (beispielsweise durch einen Buffer-Überlauf), oder ob es lediglich zu einer extremen Verlangsamung des gesamten Systems führt.

Die Sicherheitsanalyse ermöglicht das Aufdecken von möglicherweise vorhandenen Sicherheitslücken in der Software, welche sich je nach Grad der Schwere unterschiedlich äußern können; Fehlfunktion, Absturz oder unberechtigtem Zugriff. Eine solche Analyse ist vor dem Einsatz einer Serversoftware in einem produktiven Umfeld unabdingbar, damit die Sicherheit des ausführenden Systems nicht fahrlässig durch möglicherweise vorhandene Sicherheitslücken gefährdet wird.

> *„The only secure computer system in the world is unplugged, locked in a vault at the bottom of the ocean and only one person knows the location and combination of that vault. And he is dead.“* Schneier [1994]

2.1 Zuverlässigkeitsanalyse

Mit Hilfe der Zuverlässigkeitsanalyse soll herausgefunden werden, in wie weit ein CloudRun Server übermäßiger Belastung standhält und wie sich der Server in einem solchen Szenario verhält. Dabei geht es jedoch nicht um sogenannte DDoS (Distributed Denial of Service) [US-CERT] Attacken, da diese – je nach Ausmaß – die gesamte Infrastruktur lahmlegen können. Vielmehr soll untersucht werden, wie ein CloudRun Server auf möglicherweise stoßhaft auftretende, hohe Belastungen reagiert.

2.1.1 `prime` - Ein Modul zur Generierung von Last

Um ein solches Szenario realistisch darstellen zu können, wird zunächst ein CloudRun Modul benötigt, welches Last auf dem Server erzeugt. Für solche Einsatzzwecke, wo keine sinnvollen Berechnungen angestellt werden müssen, wird häufig auf die Berechnung von Primzahlen gesetzt, da dies bei ausreichend hohen Zahlen eine massive Rechenlast erzeugen kann.

Als Basis für ein solches Modul nutze ich YACOB (Yet Another CPU Overclocking Benchmark) [Pahlen, 2016], einer Software zur Erstellung von CPU-Benchmarks, im Speziellen die darin enthaltene, ausführbare Datei „`YACOBPrime.exe`“. Dieses konsolenbasierte Programm prüft Zahlen in einem angegebenen Bereich ob sie prim sind und kann dazu eine konfigurierbare Anzahl an Threads nutzen. Diese Funktionalität ermöglicht die Erzeugung künstlicher Rechenlast, um den ausführenden Computer an seine Leistungsgrenze zu bringen und somit die Stabilität eines CloudRun Servers in solchen Situationen prüfen zu können.

Um `YACOBPrime` aus CloudRun heraus nutzen zu können, muss zunächst jedoch ein Modul entwickelt werden, welches die ausführbare Datei mit den übergebenen Parametern – also „von“, „bis“ und die Anzahl der zu verwendenden Threads – aufruft.

`YACOBPrime` wird folgendermaßen gestartet, in diesem Beispiel werden alle Zahlen zwischen 1 und 100 geprüft, die Berechnung wird dabei auf zwei Threads verteilt:

```
$ YACOBPrime.exe 1 100 2
YACOB Prime started. Range: 1 - 100 with 2 Threads.
Adding task 0: 1 - 100 with 2 steps
Adding task 1: 2 - 100 with 2 steps
Finished.
```

Das CloudRun Modul muss folglich die drei Parameter entgegen nehmen und in den aufzurufenden Befehl integrieren. Das Ergebnis ist grundsätzlich nicht von Interesse, da `YACOBPrime` selbst nur eine Statusmeldung zurückgibt, nicht etwa die gefundenen Primzahlen, da es nur zu Auslastungszwecken entwickelt wurde. Die Statusmeldung wird dennoch übermittelt, um möglicherweise auftretende Fehler aufdecken zu können.

```
// Abhaengigkeiten und Vorbereitung
var exec = require('child_process').exec;
var json_error_generator = require('../app/tools/json_error_generator');
var VERSION = "1.0.0";

// Auszufuehrende Funktion
exports.run = function(data, next) {
    const convert = exec(                    // Kindprozess erzeugen
      'call "C:\\CloudRun\\YACOBPrime.exe" ' // Aufruf der EXE
          +data.from.toString()              // von
          +' '+data.to.toString()            // bis
          +' '+data.threads.toString()+'',   // Anzahl Threads
      { encoding: "latin1" }, // Enkodierung in latin1
      function(err, stdout, stderr) { // Aufzurufende Funktion wenn
                                      // Ausfuehrung beendet wurde
          if(!err) { // Wenn keine Fehler aufgetreten sind
              // Ausgabe aus stdout erzeugen
              var output =
                  { answer: Buffer.from(stdout, "ascii").toString() };

              if(output.answer==="") { // Wenn stdout leer war
                  next(json_error_generator(7)); // Fehler zurueckgeben
              }
              else { // Wenn alles korrekt verlief
                  next(output); // Ausgabe von YACOBPrime zurueckgeben
              }
          }
          else { // Wenn ein Fehler aufgetreten ist
            console.log(err); // Fehler zu Debugging-Zwecken ausgeben
              next(json_error_generator(7)); // und Fehler zurueckgeben
          }
      });
};

// Funktion zur Ausgabe der Versionsnummer
exports.version = function() {
    return VERSION;
};
```

Hierbei ist zu beachten, dass dieses Modul selbstverständlich nur zu Zwecken einer solchen Analyse verwendet werden sollte, da über eine Manipulation der Parameter weitere Befehle ausgeführt werden können. Das Modul ist daher sehr unsicher und sollte in keinem Falle auf einem produktiven Server bereitgestellt werden. Diese Thematik wird in Kapitel 2.2.6 genauer betrachtet.

Das Modul „`prime`“ lässt sich nun über die folgende Anfrage an den CloudRun Server ausführen:

```
{
    "token": "",
    "module": "prime",
    "data": {
        "from": 1,
        "to": 100,
        "threads": 2
    }
}
```

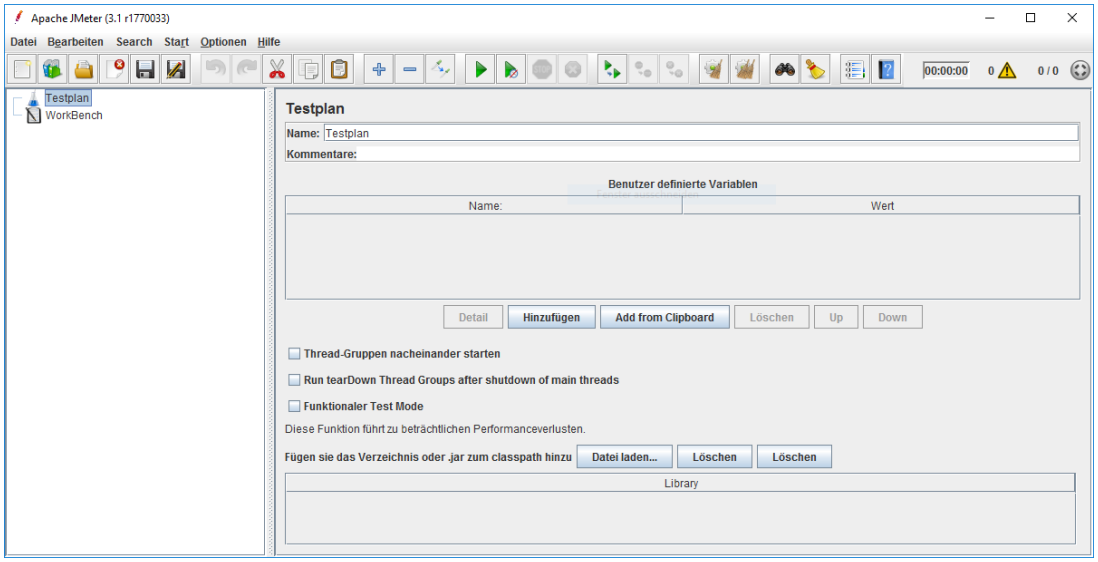
Mit Hilfe dieses Moduls kann nun geprüft werden, wie ein CloudRun Server auf eine maximale Systemauslastung reagiert. Da die Anzahl und Höhe der zu prüfenden Zahlen jedoch variabel ist, lassen sich ebenfalls Szenarien nachstellen, in denen einzelne Berechnungen weniger aufwändig sind, dafür jedoch sehr zahlreich auftreten. Die Voraussetzungen für eine Zuverlässigkeitsanalyse sind damit erfüllt.

2.1.2 Verwendete Software

Um die Auslastung des CloudRun Servers simulieren zu können, wird zunächst jedoch noch eine Software benötigt, die HTTP-Anfragen an den CloudRun Server verschickt und die Antwortzeiten des Servers gleichzeitig auswerten kann. Ich habe mich dabei für die Verwendung von „Apache JMeter“ [Apache Software Foundation, a] entschieden, da diese Software unter einer OpenSource Lizenz vertrieben wird und somit ohne Einschränkungen und finanziellen Aufwand verwendet werden kann.

Apache JMeter ist eine Software für Auslastungstests vieler verschiedener Protokolle und Systeme (HTTP, FTP, Datenbanken via JDBC, Mailserver, etc.) und eignet sich hervorragend für die geplante Zuverlässigkeitsanalyse. Stark vereinfacht sendet JMeter beliebig viele Anfragen gleichzeitig an einen Server und wartet auf dessen Antwort. Sobald eine einzelne Antwort eintrifft, wird – solange keine Abbruchbedingung erreicht ist – sofort die nächste Anfrage abgeschickt. Die Besonderheit bei JMeter ist, dass

Abbildung 2.1: Apache JMeter - Startbildschirm



sehr gründlich aufgezeichnet wird, wann eine Anfrage verschickt wurde und wann dessen Antwort eintraf, was eine Analyse der sogenannten Antwortzeiten (also die Zeitspanne zwischen Abschicken der Anfrage und Erhalten der Antwort) ermöglicht.

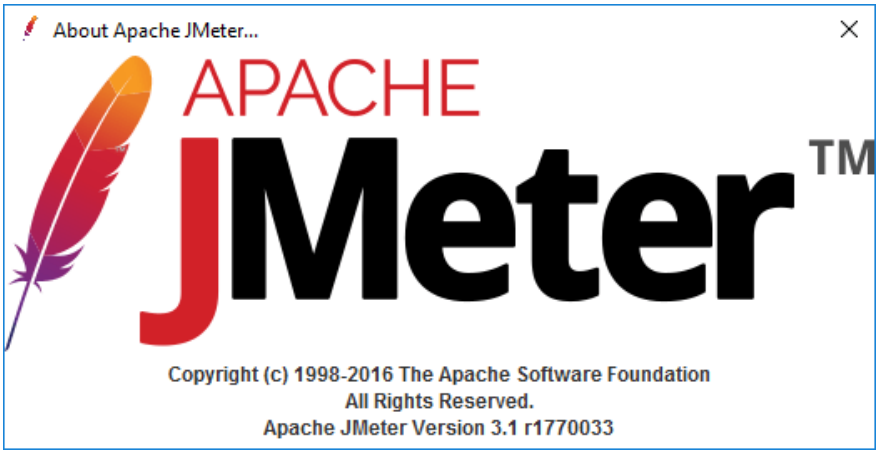
Über die Antwortzeiten lässt sich beobachten, wie ein Server in starken Belastungssituationen reagiert, beispielsweise ob einzelne Anfragen extrem langsam verarbeitet werden oder ob der Server insgesamt sehr träge reagiert.

Die Konfiguration von JMeter ist zunächst sehr kompliziert, da das Tool weitaus mehr Funktionen bereitstellt, als in dieser Zuverlässigkeitsanalyse genutzt werden können. JMeter ist sehr modular aufgebaut, um für nahezu jedes Einsatzszenario eingesetzt werden zu können, jedoch erschwert genau diese Eigenschaft die Durchführung einfacher Tests ohne sich vorher gründlich mit der Software beschäftigt zu haben.

Anwendungen wie JMeter finden fast ausschließlich Anwendung im professionellen Umfeld („Penetration Testing“), weshalb die Verfügbarkeit von frei verfügbaren Anleitungen zur Verwendung von JMeter relativ gering ausfällt. Apache selbst bietet jedoch eine sehr ausführliche Dokumentation [Apache Software Foundation, b], welche den gesamten Funktionsumfang von JMeter gründlich erläutert.

Zur Durchführung dieser Zuverlässigkeitsanalyse habe ich Apache JMeter in der Version `3.1 r1770033` verwendet (Abbildung 2.2).

Abbildung 2.2: Apache JMeter - Über JMeter



2.1.3 Testumgebung

Grundsätzlich sind absolute Rechenzeiten für eine Zuverlässigkeitsanalyse eher uninteressant, jedoch ist es sinnvoll eine mögliche Verfälschung der Ergebnisse so weit wie möglich auszuschließen. Daher lief CloudRun zur Durchführung der Zuverlässigkeitsanalyse auf einem eigenen Computer, um die Ergebnisse nicht durch die Belastung des Systems durch Apache JMeter zu beeinflussen. Der ausführende Computer verfügt über die folgende Hard- und Softwarekonfiguration:

```
Betriebssystem: Microsoft Windows 10 Pro - 10.0.14393.206
NodeJS-Version: NodeJS v7.7.2

CPU:            AMD FX-8350, 4C/8T, Uebertaktung auf 4.8GHz
Motherboard:    ASUS M5A97 R2.0 evo
RAM:            12GB, DDR3 1600MHz CL8, Dual Channel
Speicher:       Crucial CT256MX100SSD1 (SSD)
```

Angesprochen wird der CloudRun Server über einen zweiten Computer, welcher über Gigabit-Ethernet mit dem ausführenden Computer verbunden ist.

CloudRun wurde für die Durchführung der Zuverlässigkeitsanalyse so konfiguriert, dass keine Überprüfung des Authentifizierungstokens vorgenommen wird.

2.1.4 Maximale Systemauslastung

Da nun alle Vorbereitungen zur Durchführung der Zuverlässigkeitsanalyse getroffen sind, kann mit dem ersten Testszenario begonnen werden. In diesem Fall soll zunächst

geprüft werden, wie CloudRun auf eine maximale Systemauslastung reagiert.

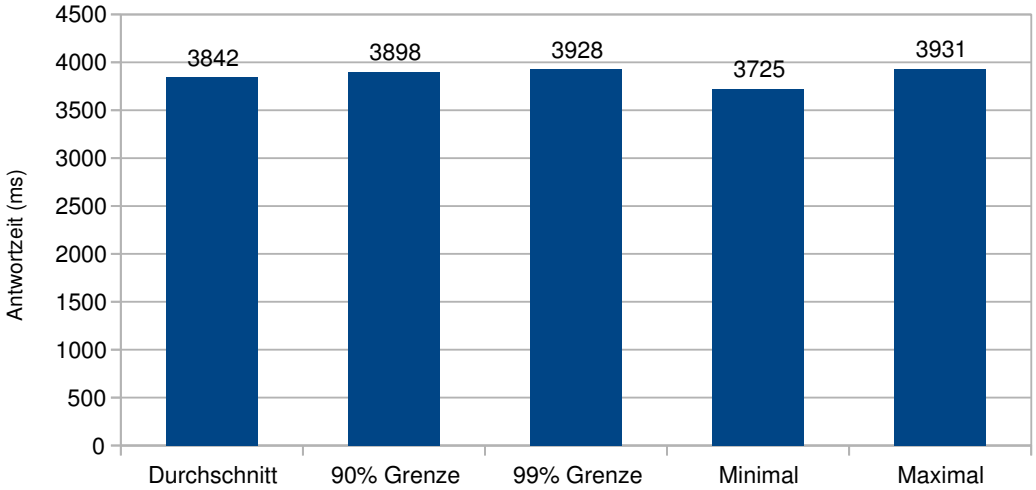
Vorstellbar wäre, dass CloudRun in einem solchen Szenario extrem träge wird und nicht mehr ordentlich auf eintreffende Anfragen reagiert. Alternativ wäre es auch möglich, dass CloudRun selbst vollständig abstürzt. Ein solches Verhalten wäre allerdings eher auf NodeJS zurückzuführen, da dies die Ressourcenverwaltung übernimmt.

In diesem Fall sollen alle Zahlen zwischen 1 und 3.000.000 untersucht werden, pro Anfrage werden dafür 4 Threads verwendet. Insgesamt wurden drei Tests mit verschiedenen Einstellungen durchgeführt.

Einzustellen ist die Anzahl der Threads von JMeter, also wie viele Anfragen zeitgleich verschickt werden sollen sowie die Anzahl der Wiederholungen. Eine Konfiguration von beispielsweise 10 Threads und 15 Wiederholungen resultiert in insgesamt 150 Anfragen. Zusätzlich lässt sich die sogenannte „Ramp-Up-Time“ einstellen, diese Option sorgt dafür dass anfangs nicht alle Threads zeitgleich sondern leicht versetzt gestartet werden.

Im ersten Test werden 3 Threads mit 50 Wiederholungen gestartet, die Ramp-Up-Time beträgt eine Sekunde. Die Last für den Server ist dabei noch relativ gering, da durch das Modul „prime“ insgesamt nur 12 Threads zur Berechnung gestartet werden.

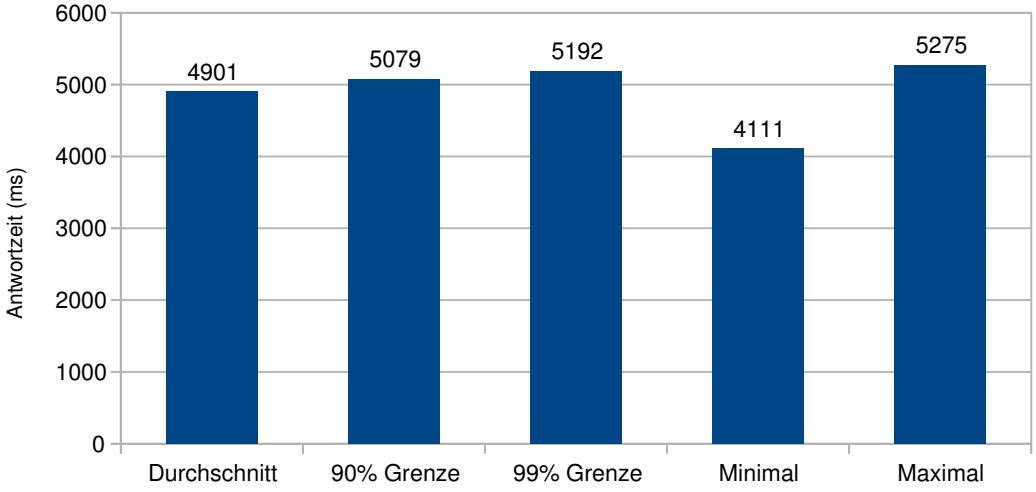
Abbildung 2.3: Antwortzeiten - 3 Threads / 50 Wiederholungen



Die grafische Darstellung der Antwortzeiten in diesem Test (Abbildung 2.3) zeigt deutlich, dass die Belastung zu keiner Beeinflussung der Geschwindigkeit des CloudRun Servers geführt hat. Zu erkennen ist dies daran, dass alle Werte dicht beieinander liegen und somit alle Anfragen gleichwertig abgearbeitet wurden. Der Wert „90% Grenze“ beschreibt, dass 90% der abgeschickten Anfragen in der angegebenen Antwortzeit bearbeitet wurden. Dies gilt dementsprechend ebenso für den Wert „99% Grenze“. Mit Hilfe dieser Werte lassen sich Ausreißer identifizieren – sollte also zwischen der „99% Grenze“ und dem Maximalwert eine große Differenz liegen, ist davon auszugehen, dass nur wenige, einzelne Anfragen für den erhöhten Maximalwert verantwortlich sind.

Um die Belastung auf den Server zu steigern wird der Test nun mit 5 Threads und 30 Wiederholungen durchgeführt.

Abbildung 2.4: Antwortzeiten - 5 Threads / 30 Wiederholungen

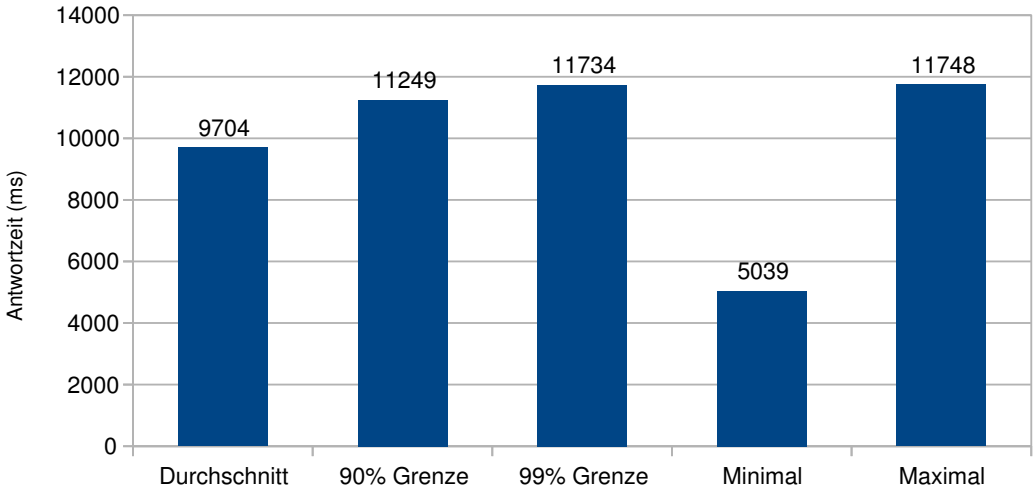


Dabei ist zu erkennen (Abbildung 2.4), dass nun größere Differenzen in den Antwortzeiten auftreten. Insbesondere die Differenz zwischen der minimalen und der maximalen Antwortzeit lässt darauf schließen, dass zumindest einzelne Anfragen wesentlich schneller verarbeitet wurden. Da der Maximalwert jedoch nicht nennenswert von der durchschnittlichen Antwortzeit abweicht, kann davon ausgegangen werden, dass ein Großteil der Anfragen mit einer ähnlichen Antwortzeit verarbeitet wurden.

Als letzte Steigerung wird der Test mit 10 Threads und 15 Wiederholungen durchgeführt. Zu erwarten ist eine wachsende Differenz zwischen Minimal- und

Maximalantwortzeiten, da nun insgesamt 40 Threads zur Berechnung der Zahlen auf dem Server gestartet werden.

Abbildung 2.5: Antwortzeiten - 10 Threads / 15 Wiederholungen



Wie erwartet ist die Differenz zwischen Minimal- und Maximalantwortzeit stark gestiegen. Zudem ist nun zu erkennen, dass auch die Differenz zwischen der durchschnittlichen Antwortzeit und der maximalen Antwortzeit größer wird. Dies bedeutet, dass die Antwortzeiten in diesem Test erheblich schwanken, was sich zusätzlich durch eine grafische Darstellung der einzelnen Antwortzeiten als Liniendiagramm darstellen lässt (Abbildung 2.6).

Interessant ist dabei, dass die Anfragen anfangs relativ gleichwertig behandelt werden, jedoch ab der ungefähr 100. Anfrage erhebliche Schwankungen in der Antwortzeit auftreten. Um dieses Verhalten zu Überprüfen, wurde der Test erneut durchgeführt, jedoch mit 30 statt 15 Wiederholungen, um die Dauer des Tests zu erhöhen (Abbildung 2.7).

In diesem Falle treten über die gesamte Dauer des Tests stärkere Schwankungen auf. Eine genaue Erklärung für dieses Phänomen zu nennen ist schwierig, da zu viele äußere Umstände zu einem solchen Verhalten führen können. Möglicherweise beeinflusst ein aktiver Hintergrundprozess die Ausführung der Threads und verursacht so eine solche Streuung der Antwortzeiten.

Letztendlich konnte jedoch bewiesen werden, dass CloudRun keine Probleme mit einer maximalen Systemauslastung hat. Die Annahme, dass eine solche Belastung zu einem Absturz CloudRuns führen könnte, wurde nicht bestätigt. Zudem konnte festgestellt werden, dass keine einzelnen Anfragen während einer solch starken Belastungsphase vernachlässigt werden, sondern die Antwortzeiten innerhalb einer gewissen Toleranz relativ konstant blieben.

Abbildung 2.6: Antwortzeiten als Liniendiagramm - 10 Threads / 15 Wiederholungen

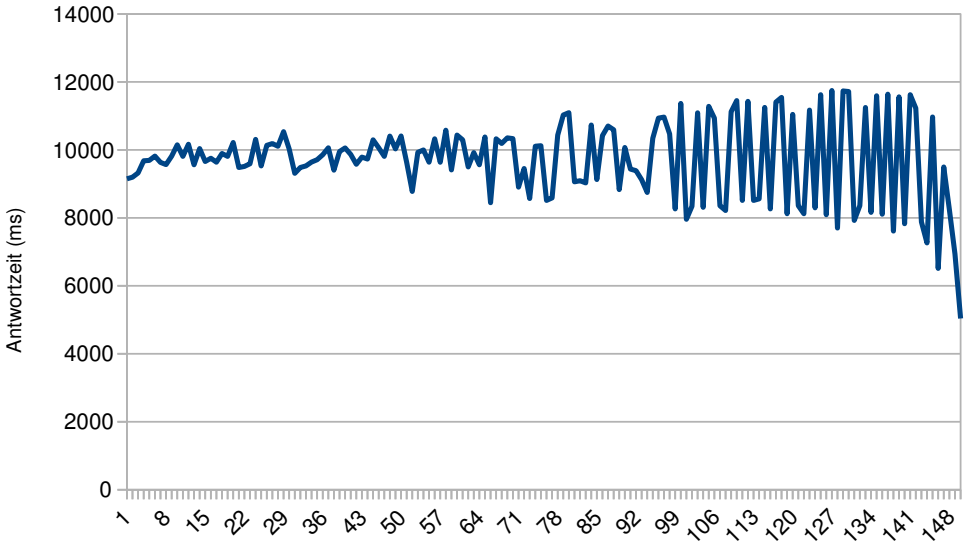
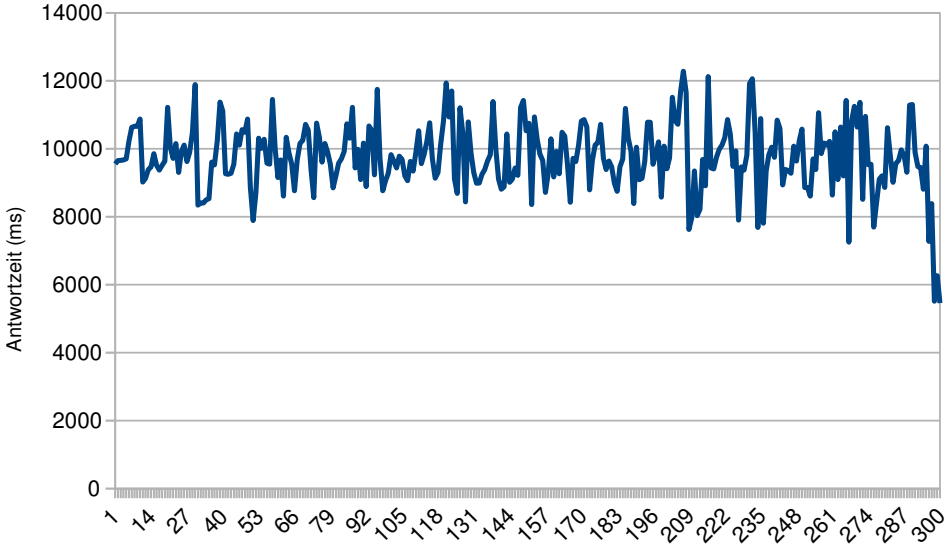


Abbildung 2.7: Antwortzeiten als Liniendiagramm - 10 Threads / 30 Wiederholungen



Abschließend folgt eine tabellarische Zusammenfassung der Messdaten zum Vergleich der einzelnen Tests (Tabelle 2.1). Zusätzlich ist hierbei die Laufzeit der einzelnen Tests vermerkt. Dabei fällt bei den ersten drei Tests, welche alle insgesamt 150 Anfragen verschickten, auf, dass die Effizienz mit einer höheren Anzahl an gleichzeitigen Anfragen gesteigert (also die Laufzeit verkürzt) wird.

Tabelle 2.1: Zusammenfassung der Messdaten - Maximale Systemauslastung

Thr.	**Wiederh.**	**Durchschn.**	**90%**	**Min.**	**Max.**	**Laufzeit**
3	50	3.842ms	3.898ms	3.725ms	3.931ms	189s
5	30	4.901ms	5.079ms	4.111ms	5.275ms	145s
10	15	9.704ms	11.249ms	5.039ms	11.748ms	143s
10	30	9.758ms	10.889ms	5.458ms	12.280ms	290s

2.1.5 Anfragen-Überschwemmung

Bisher wurde geprüft, wie CloudRun auf gleichzeitig auftretende, rechenintensive Belastungen reagiert. Ein weiteres mögliches Szenario ist jedoch das sogenannte Flooding (Überschwemmen), bei dem extrem viele, kleine Aufgaben eintreffen. Dies kann entweder der Fall sein, wenn tatsächlich sehr viele Aufgaben berechnet werden müssen, jedoch wird diese Methode häufig als bösartiger Angriff auf eine Serversoftware eingesetzt, um sie beispielsweise zum Absturz zu bringen.

Da bei dieser Art der Belastung die einzelnen Anfragen schnell verarbeitet werden können, werden pro Anfrage die Zahlen 1 bis 15.000 überprüft, dabei werden nur 2 Threads eingesetzt.

Im ersten Test werden 50 Threads mit 50 Wiederholungen ausgeführt, insgesamt also 2.500 Anfragen verschickt. Die Ramp-Up-Time ist von nun an deaktiviert, um ein schlagartiges Eintreffen vieler, kleiner Anfragen zu simulieren (Abbildung 2.8).

Es fällt sofort die sehr große Differenz zwischen minimaler und maximaler Antwortzeit auf. Betrachtet man die Antwortzeiten jedoch als Liniendiagramm, wird deutlich, dass diese große Differenz auf Grund des Abklingens zum Ende des Tests auftritt (Abbildung 2.9). Aus diesem Grund sollten die Minimalwerte bei dieser Art von Belastung nicht weiter beachtet werden.

Abbildung 2.8: Antwortzeiten - 50 Threads / 50 Wiederholungen

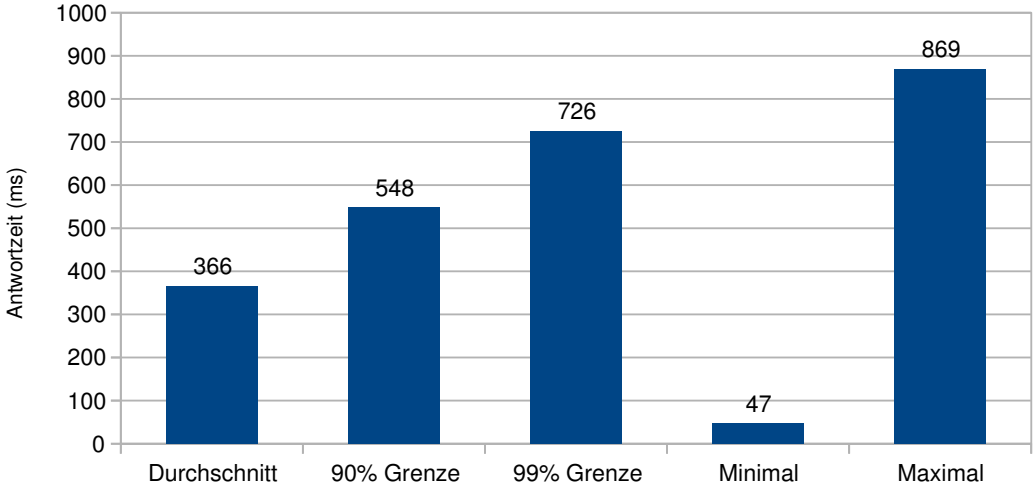
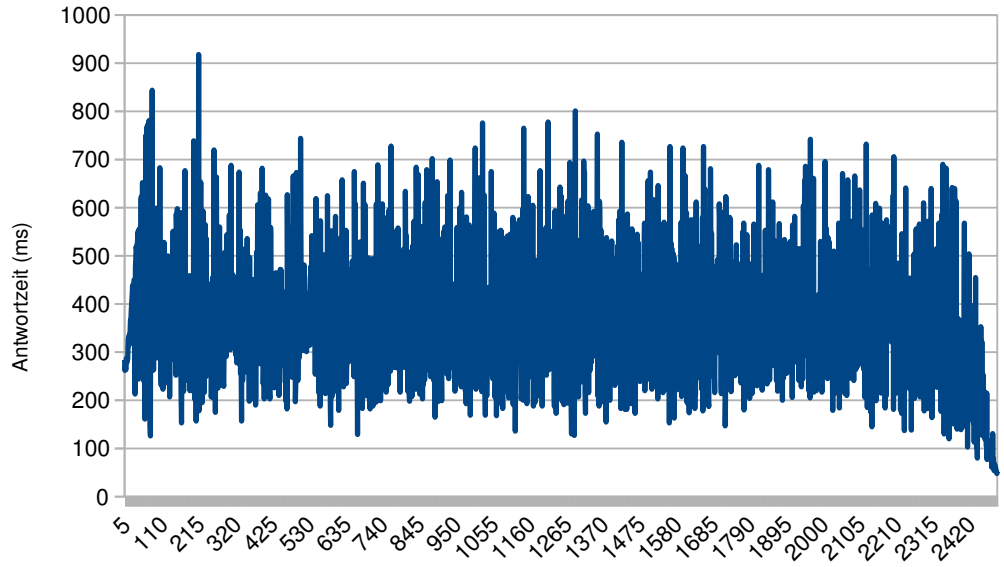


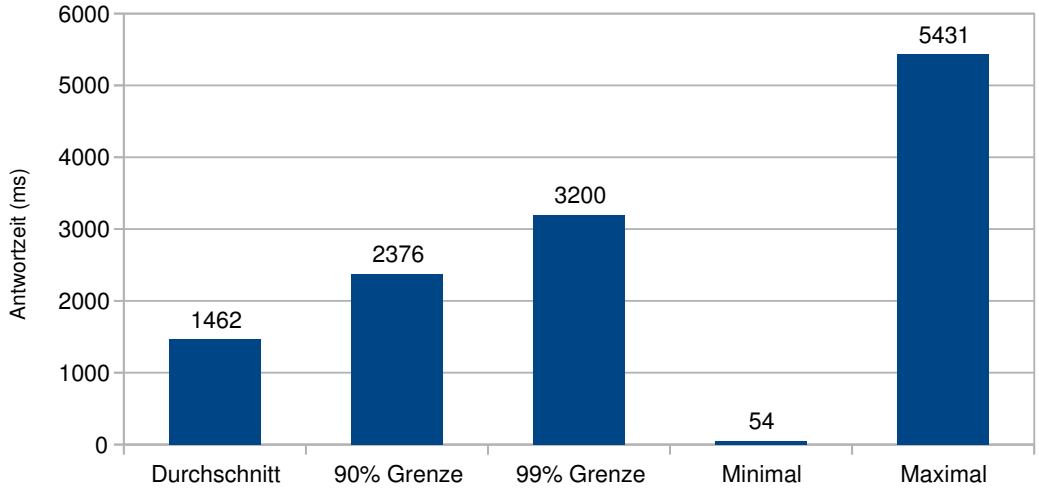
Abbildung 2.9: Antwortzeiten als Liniendiagramm - 50 Threads / 50 Wiederholungen



Im nächsten Test soll die Belastung deutlich gesteigert werden, da der CloudRun Server sehr zuverlässig auf den vorhergehenden Test reagiert hat. Zwar konnte durch die hohe Belastung eine deutlich gesteigerte, durchschnittliche Antwortzeit im Verhältnis zu einer einzelnen Anfrage (vgl. Minimalwert) erreicht werden, jedoch kann dabei nicht von einer Unzuverlässigkeit gesprochen werden. Im folgenden Test wurden 200 Threads bei 50 Wiederholungen gestartet, es wurden insgesamt also

10.000 Anfragen verschickt.

Abbildung 2.10: Antwortzeiten - 200 Threads / 50 Wiederholungen



Auch in diesem Fall reagiert CloudRun zuverlässig, wenn auch in einzelnen Fällen (vgl. Maximalwert) deutlich verzögert. In diesem Falle ist die 90% sowie 99% Grenze von Interesse, da ein Großteil der Anfragen mit einer akzeptablen Antwortzeit verarbeitet wird.

Um die Belastung weiter zu steigern, wird der nächste Test mit 500 Threads bei 50 Wiederholungen ausgeführt, also insgesamt 25.000 Anfragen (Abbildung 2.11). Selbst bei einer solch starken Belastung reagiert der CloudRun Server souverän. Zwar ist die durchschnittliche Antwortzeit auf 3.739ms gestiegen, dies entspricht aber nur einem Faktor von etwa 65 im Vergleich zur Antwortzeit einer einzelnen Anfrage (vgl. Minimalwert).

Auch für die Messdaten der Anfragen-Überschwemmung folgt eine tabellarische Zusammenfassung (Tabelle 2.2).

Während der Durchführung der Überschwemmungs-Tests fiel auf, dass CloudRun ab einem bestimmten Punkt keine Einträge in der Log-Datei anlegt. Dieses Verhalten wird ebenso mit einer Fehlermeldung in der Konsole bestätigt (Abbildung 2.12).

Abbildung 2.11: Antwortzeiten - 500 Threads / 50 Wiederholungen

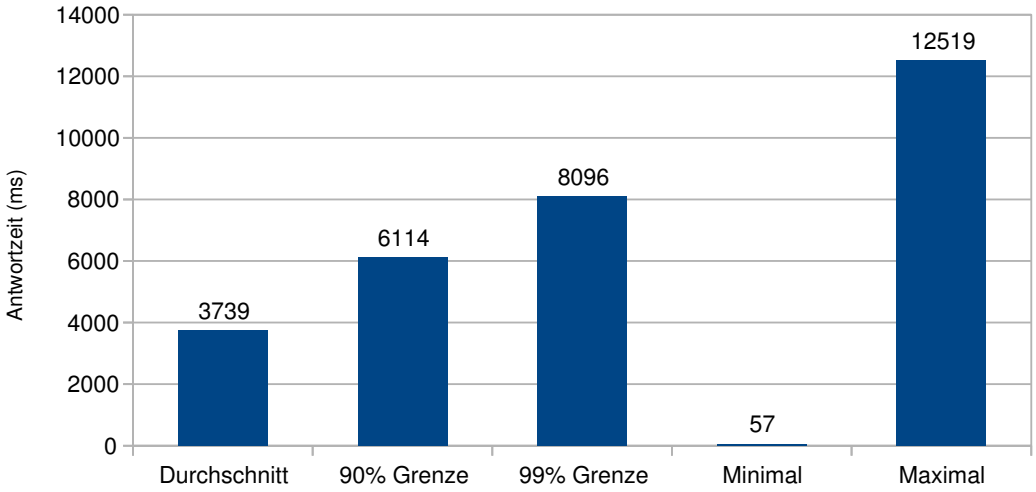
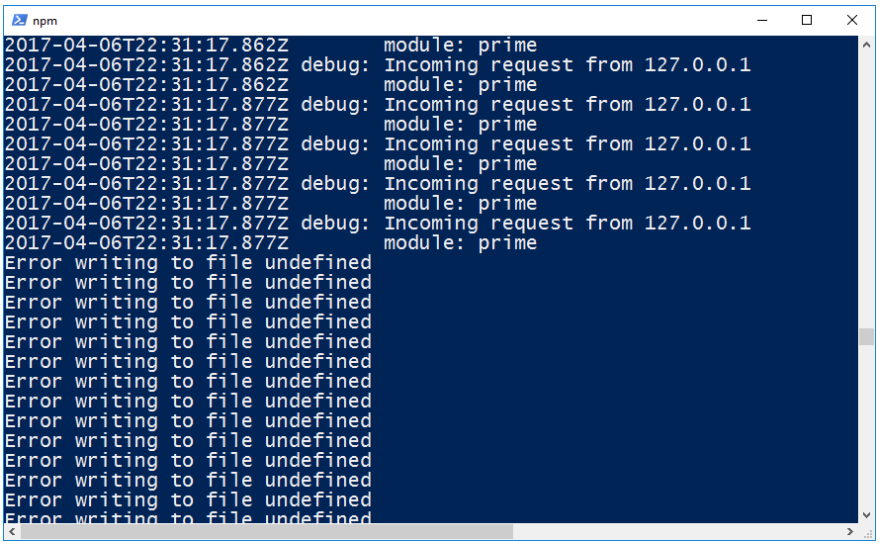


Tabelle 2.2: Zusammenfassung der Messdaten - Anfragen-Überschwemmung

Thr.	Wiederh.	Durchschn.	90%	Min.	Max.	Laufzeit
50	50	366ms	548ms	47ms	869ms	19s
200	50	1.462ms	2.376ms	54ms	5.431ms	76s
500	50	3.739ms	6.114ms	57ms	12.519ms	193s

Abbildung 2.12: Fehlermeldung in der Konsole unter Windows



Auffallend ist jedoch, dass dieser Fehler nur unter dem Betriebssystem Windows auftritt, da der Fehler unter Linux nicht zu reproduzieren war. Dies lässt darauf

schließen, dass in diesem Falle möglicherweise eine Fehlfunktion in NodeJS für Windows oder der PowerShell vorliegt, CloudRun dafür also nicht verantwortlich ist.

2.1.6 Evaluierung der Ergebnisse

Zusammengefasst lieferte die Zuverlässigkeitsanalyse sehr zufriedenstellende Ergebnisse. CloudRun erwies sich als sehr stabile Software, da sie trotz extremer Belastungen nicht zu Abstürzen oder Fehlfunktionen neigt. Zudem konnte festgestellt werden, dass alle eingehenden Anfragen sehr gleichwertig verarbeitet werden, weshalb im produktiven Einsatz keine übermäßig lange Wartezeiten entstehen sollten, selbst wenn der Server stark ausgelastet ist.

Ein Großteil dieser Zuverlässigkeit in Szenarien mit extrem hoher Belastung ist dabei NodeJS zuzuschreiben, da CloudRun nicht explizit für solche Szenarien optimiert worden ist. NodeJS basiert auf der sehr robusten „Chrome V8“ JavaScript Engine [Google], welche für ihren sehr zuverlässigen Garbage-Collector bekannt ist. Das bedeutet, dass vorhandene Objekte – sobald sie nicht mehr verwendet werden – automatisch aus dem Systemspeicher entfernt werden. Dies führt zu einer geringen Belastung des Arbeitsspeichers und ermöglicht so das Verarbeiten von extrem vielen, eingehenden Anfragen, ohne dabei in Stabilitätsprobleme zu geraten.

Die durchgeführte Zuverlässigkeitsanalyse ist ein erster Anhaltspunkt für die Stabilität und Zuverlässigkeit CloudRuns. Grundsätzlich lassen sich solche Analysen noch viel detaillierter durchführen, jedoch mangelt es bei dieser Zuverlässigkeitsanalyse dabei an geeigneter Hardware. Apache JMeter erzeugte bei der Durchführung des 500 Thread-Tests eine erhebliche Systemlast, weshalb ich mich dagegen entschieden habe einen weiteren Tests mit noch mehr Threads durchzuführen. Dies hätte dazu führen können, dass JMeter in einen Flaschenhals gerät, welcher das Testergebnis massiv verfälscht hätte, weshalb die Ergebnisse eines solchen Tests ohnehin unbrauchbar gewesen wären. Zudem ist - abgesehen von einem Angriffsszenario (DDoS) - eine Belastung von 500 gleichzeitig eingehenden Anfragen bereits als unwahrscheinlich anzusehen, weshalb ein weiterer Test auch wenig zur Betrachtung realistischer Szenarien beigetragen hätte.

Wird CloudRun auf einem ordentlichen Server ausgeführt, also einem Computersystem mit mehr CPU-Kernen, kann von einer noch höheren Belastungsgrenze ausgegangen werden. Zudem begünstigen zusätzliche CPU-Kerne die gleichzeitige

Bearbeitung von Anfragen, was in realistischen Situationen dazu führen sollte, dass bei mittlerer Belastung des Servers keine nennenswert erhöhten Antwortzeiten zu erwarten sind, da alle Prozesse parallel ausgeführt werden können.

2.2 Sicherheitsanalyse

Da nun bekannt ist, wie ein CloudRun Server auf unterschiedliche Belastungen reagiert, kann anschließend die Sicherheitsanalyse durchgeführt werden. Diese Analyse dient der Aufdeckung von Sicherheitsrisiken, welche im produktiven Einsatz im Falle eines böswilligen Angriffes zu einer Gefährdung des gesamten Systems, oder sogar Netzwerkes führen könnten.

Da CloudRun als REST-API einen – im Verhältnis zu anderer Software – sehr begrenzten Funktionsumfang hat, sind auch die Möglichkeiten der Sicherheitsanalyse begrenzt. Jedoch gibt es einige Einfallstore für Angreifer, welche vor dem Einsatz CloudRuns in einem produktiven Umfeld unbedingt überprüft werden sollten. Die zu untersuchenden Bereiche werden im Folgenden detailliert erläutert und analysiert.

Disclaimer Alle in diesem Kapitel genannten Möglichkeiten zur Überprüfung einer Software auf Sicherheitsrisiken dienen ausschließlich dem Zweck einer solchen Sicherheitsanalyse. Ich übernehme keine Haftung für möglicherweise entstandene Schäden bei Nachahmung der Analyse und weise ausdrücklich darauf hin, dass die hier verwendete Software ausschließlich für analytische Zwecke eingesetzt werden darf.

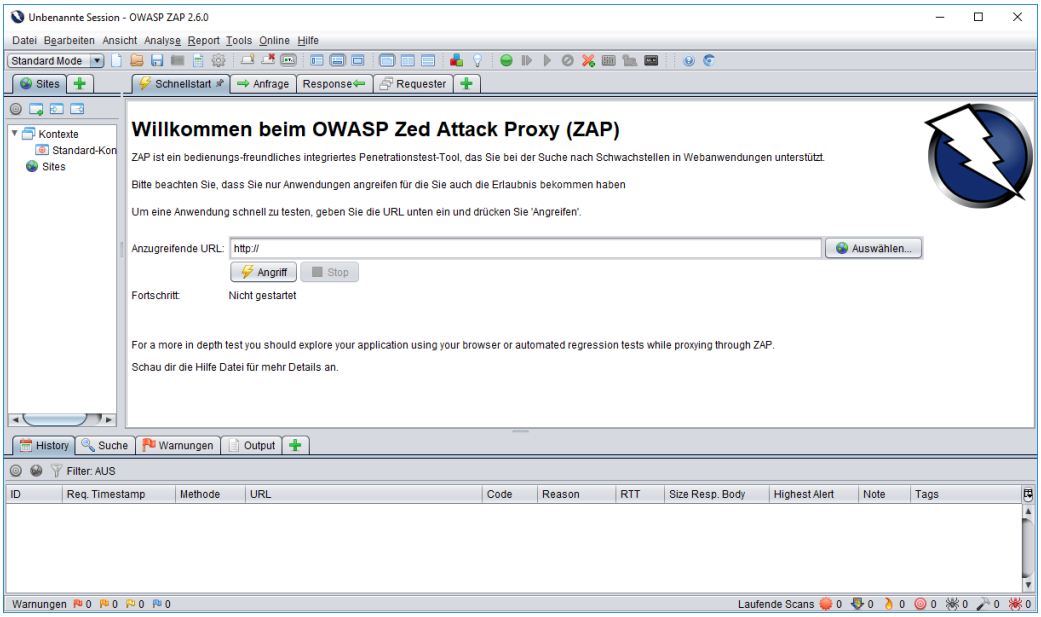
2.2.1 Verwendete Software

OWASP ZAP

Für die Durchführung dieser Sicherheitsanalyse habe ich mich für die quelloffene Software „OWASP ZAP“ [OWASP, b] entschieden. ZAP (Zed Attack Proxy) ist eine Software, welche HTTP-Anfragen als Proxy mitschreiben und manipulieren kann und ermöglicht dadurch die Sicherheitsanalyse einer Serveranwendung anhand der gestellten Anfragen.

ZAP ist vergleichbar mit der bekannten Software „Wireshark“, welche jedoch den gesamten Netzwerktraffic mitschneidet. ZAP hingegen wird als HTTP-Proxy konfiguriert, sodass die gesendeten Anfragen mitgeschnitten werden können. Die

Abbildung 2.13: OWASP ZAP - Startbildschirm



aufgezeichneten Anfragen lassen sich anschließend betrachten und wahlweise manuell oder mit Hilfe implementierter Tools gezielt manipulieren.

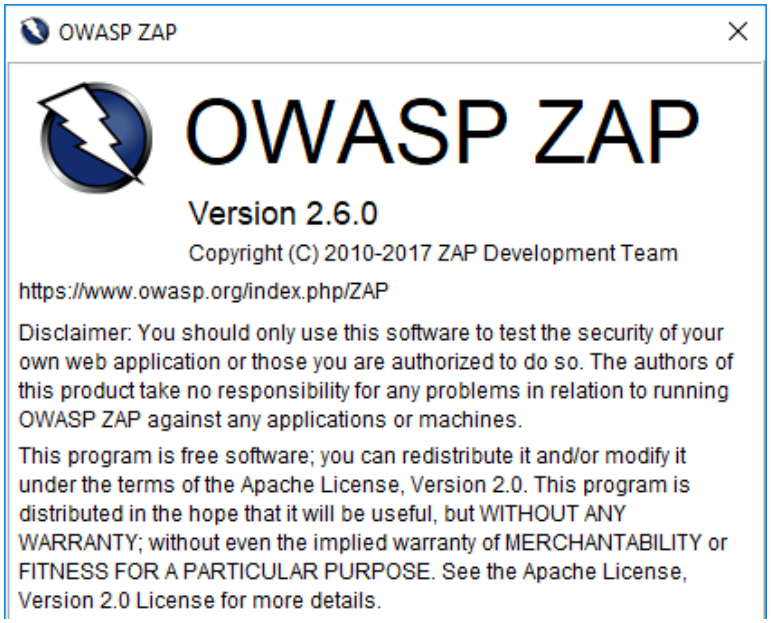
Für diese Sicherheitsanalyse habe ich OWASP ZAP in der Version 2.6.0 verwendet (Abbildung 2.14).

Neben ZAP gibt es eine Vielzahl an vergleichbaren Programmen, beispielsweise „IBM Security AppScan“ oder „Burp Suite“ [sic!] von Portswigger. Letztere ist zwar in einer kostenlosen Variante verfügbar [Portswigger], jedoch ist diese stark im Funktionsumfang beschnitten und daher für die geplanten Teilanalysen ungeeignet.

sqlmap

Da CloudRun standardmäßig eine Unterstützung zur Token-Authentifizierung per MySQL-Datenbank liefert, sollte die Sicherheit dieser Implementation ebenfalls getestet werden. Bei Anwendungen, welche Verbindungen zu Datenbanken herstellen besteht grundsätzlich das Risiko sogenannter SQL-Injections (genaueres dazu in Kapitel 2.2.5), welche mit entsprechenden Kenntnissen zwar auch manuell gefunden werden können, jedoch bietet die Software „sqlmap“ [sqlmap Projektseite] die Möglichkeit, solche Sicherheitsrisiken vollautomatisiert aufzudecken.

Abbildung 2.14: OWASP ZAP - Über ZAP (Ausschnitt)



Des Weiteren bietet sqlmap die Möglichkeit, sobald ein Angriffspunkt der SQL-Injection gefunden wurde, das verwendete Datenbanksystem und dessen Struktur zu analysieren. Dies ist gleichzeitig ein Indikator dafür, wie gefährlich SQL-Injections sind.

sqlmap ist eine für Python 2.7 geschriebene, quelloffene und konsolenbasierte Software, welche ich für diese Sicherheitsanalyse in der Version 1.1.4-11 verwendet habe.

2.2.2 Testumgebung

Für die Sicherheitsanalyse wird CloudRun auf dem gleichen System ausgeführt, welches in Kapitel 2.1.3 beschrieben wurde. In diesem Falle wird die Analysesoftware jedoch ebenfalls auf diesem System ausgeführt, da die Antwortzeiten des CloudRun Servers bei einer Sicherheitsanalyse nicht relevant sind.

CloudRun wird für die Sicherheitsanalyse jedoch so konfiguriert, dass die Token-Authentifizierung über das Authentifizierungsmodul `textfile_auth` abgewickelt wird, der zugelassene Token also in einer Textdatei aufgelistet ist. Dadurch ist es möglich, darin enthaltene Sicherheitsrisiken aufzudecken. Eine Analyse des Authentifizierungsmoduls `mysql_auth` findet gesondert in Kapitel 2.2.5 statt.

Für die Sicherheitsanalyse wird das standardmäßig enthaltene Modul `stringAnalyzer` verwendet, da keine Rechenlast erzeugt werden muss.

2.2.3 Überprüfen auf verbreitete Sicherheitsrisiken

OWASP ZAP bietet die Möglichkeit, einen Webserver sowie den darauf laufenden Dienst auf verbreitete Sicherheitsrisiken zu überprüfen („Aktiver Scan“). Dieser Test ist sehr allgemein, weshalb er sich gut als Einstiegspunkt einer Sicherheitsanalyse eignet. Tabelle 2.3 beschreibt die dabei durchgeführten Angriffe [OWASP, a].

Zur Durchführung dieses Tests wurde ZAP auf die aggressivste Stufe („Irre“) eingestellt. Angegriffen wurden dabei die aufgerufene URL, die über den POST-Request übertragenen Daten, den URL-Pfad sowie die HTTP-Header. Zur Durchführung des Tests wurden 2012 Anfragen an den CloudRun Server verschickt (Abbildung 2.15).

Abbildung 2.15: OWASP ZAP - Aktiver Scan

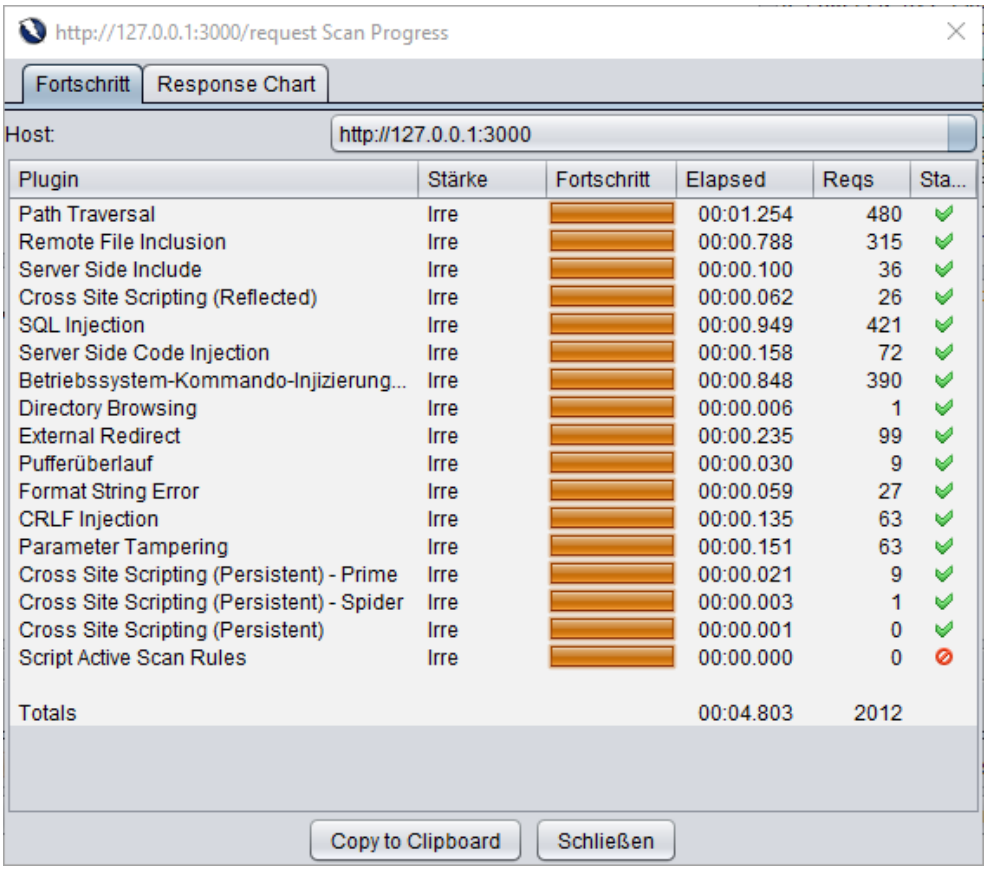


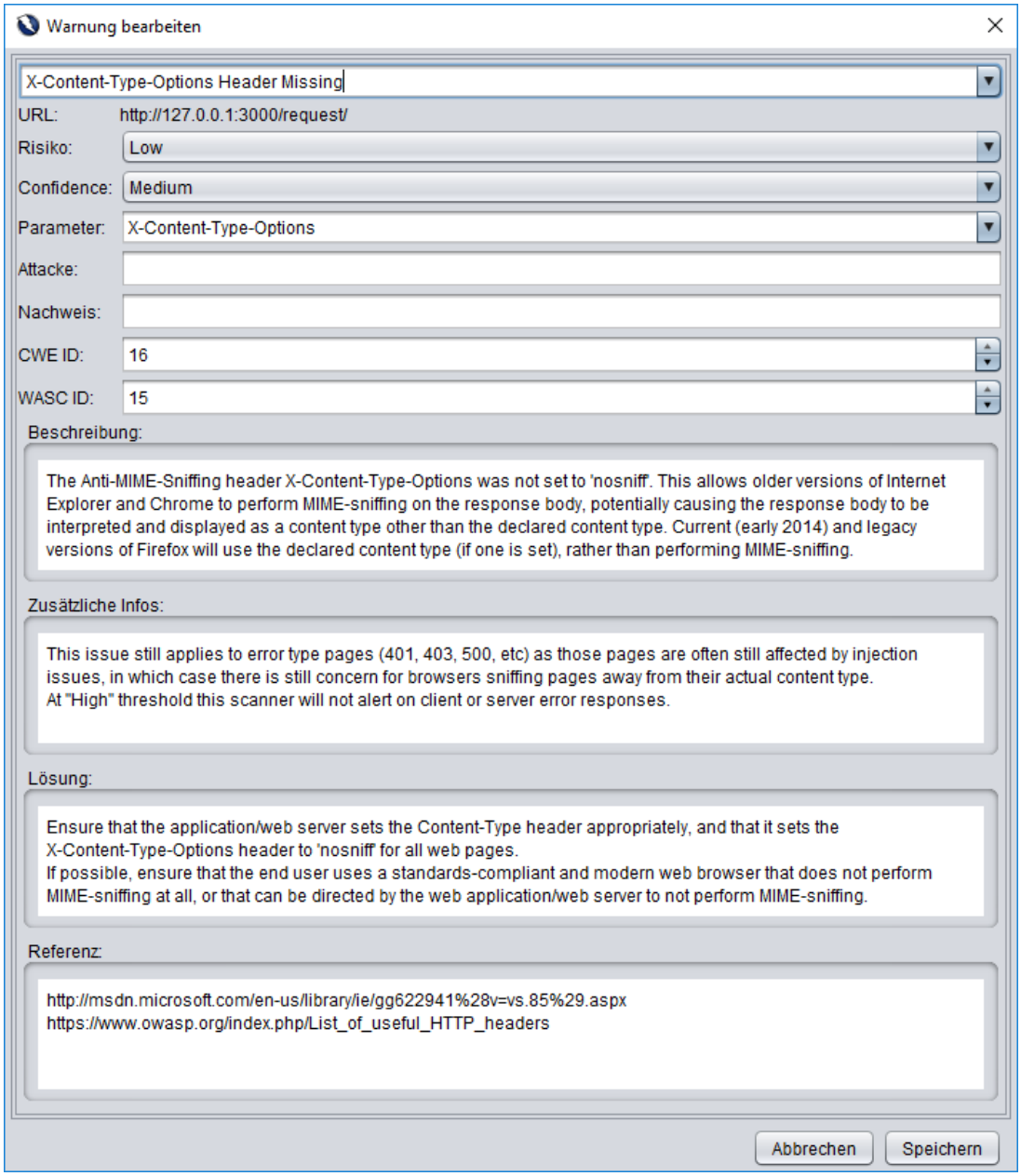
Tabelle 2.3: OWASP ZAP: Auflistung der Tests im Aktiven Scan

Name	Beschreibung
Path Traversal	Ressourcen außerhalb des Web-Verzeichnisses abrufen, beispielsweise `/etc/passwd`
Remote File Inclusion	Externe Dateien in den Code einschleusen, um ihn auszuführen
Server Side Include	Externe Dateien per Server Side Include einschleusen
SQL Injection	Ausführen von SQL Kommandos
Server Side Code Injection	Codebefehle einschleusen und ausführen
Betriebssystem-Kommando-Injizierung	Ausführen von Kommandozeilenbefehlen
Directory Browsing	Rekonstruktion der Ordnerstruktur
External Redirect	Manipulation von Redirects, um externe Dateien abzurufen
Pufferüberlauf	Ausführung von Kommandos mittels Pufferüberlauf
Format String Error	Provozieren von Fehlermeldungen durch Übermittlung besonderer Zeichen
CRLF Injection	Manipulation von HTTP-Header-Daten durch Einsetzen von expliziten Zeilenumbrüchen (Carriage Return, Line Feed)
Parameter Tampering	Manipulation von übertragenen Parametern

Der aktive Scan hat keine Sicherheitslücken entdecken können, jedoch wurde eine Warnung ausgegeben, welche ebenfalls ein potenzielles Sicherheitsrisiko darstellen könnte (Abbildung 2.16). CloudRun sollte dahingehend so konfiguriert werden, dass der HTTP-Header „`X-Content-Type-Options`“ permanent auf „`nosniff`“ gesetzt wird.

Zur Verifizierung der Warnung habe ich den `request`-Handler des CloudRun Servers um diese HTTP-Header Angabe ergänzt und den aktiven Scan erneut durchgeführt.

Abbildung 2.16: OWASP ZAP - Aktiver Scan - Gefundene Warnung



Die Warnung konnte nicht erneut erzeugt werden, das Sicherheitsrisiko wurde daher behoben.

Da neben dieser Warnung keine weiteren Risiken gefunden wurden, kann der Test als abgeschlossen angesehen werden. Es wird jedoch bereits jetzt deutlich, dass

eine solche Sicherheitsanalyse unabdingbar ist, da solche Sicherheitsrisiken ohne die Verwendung entsprechender Spezialsoftware kaum aufzufinden sind. Zwar handelte es sich dabei nicht um ein schwerwiegendes Risiko, jedoch sollte eine Software so weit wie möglich abgesichert werden, bevor sie produktiv eingesetzt wird.

2.2.4 Anfragen-Fuzzing

Der aktive Scan von OWASP ZAP ist sehr allgemein gehalten, damit er möglichst schnell durchläuft. Um weitere Fehler aufdecken zu können, wendet man das sogenannte „Fuzzing“ an. Dabei werden die übertragenen Parameter (in diesem Falle im JSON-Format) automatisch angepasst. Als Vorlagen für diese Anpassungen verwendet man dabei Dateilisten, dessen Einträge bewusst Fehlfunktionen provozieren.

Ich verwende als Vorlage für diesen Test die „FuzzDB“ [github/fuzzdb-project], welche bekannte Angriffsziele enthält und somit möglichst genaue Ergebnisse liefern kann. Dabei ist es nicht empfehlenswert, alle Einträge einer solchen Dateiliste zu verwenden, da dadurch mitunter mehrere Hunderttausend Anfragen generiert werden können.

Vielmehr muss eine Selektion vorgenommen werden, denn große Teile der FuzzDB werden offensichtlich keinen Effekt haben (beispielsweise Listen häufig verwendeter Passwörter). Tabelle 2.4 beschreibt die von mir verwendeten Listen.

Tabelle 2.4: OWASP ZAP: Verwendete Fuzzing-Listen

Name	Beschreibung
attack / all-attacks	Bekannte Strings um Fehler in Betriebssystemen auszulösen
attack / control-chars	Alle Kontrollzeichen (bspw. Zeilenumbruch) um möglicherweise Fehler auszulösen
attack / format-strings	Bekannte Strings um Fehler zur falschen Formatierung von Strings auszulösen
attack / json	Bekannte Strings um Fehler bei JSON-Übertragungen auszulösen

Betrachtet man nun eine Anfrage an den CloudRun Server, muss entschieden werden, an welchen Stellen das Fuzzing angwendet werden soll. In diesem Falle ist es

sinnlos, den `data`-Parameter zu fuzzen, da dieser ausschließlich vom Modul verarbeitet wird. Sinn der Sicherheitsanalyse ist jedoch nicht die Sicherheit von einzelnen Modulen zu überprüfen (dazu mehr in Kapitel 2.2.6), sondern vielmehr den CloudRun Server an sich.

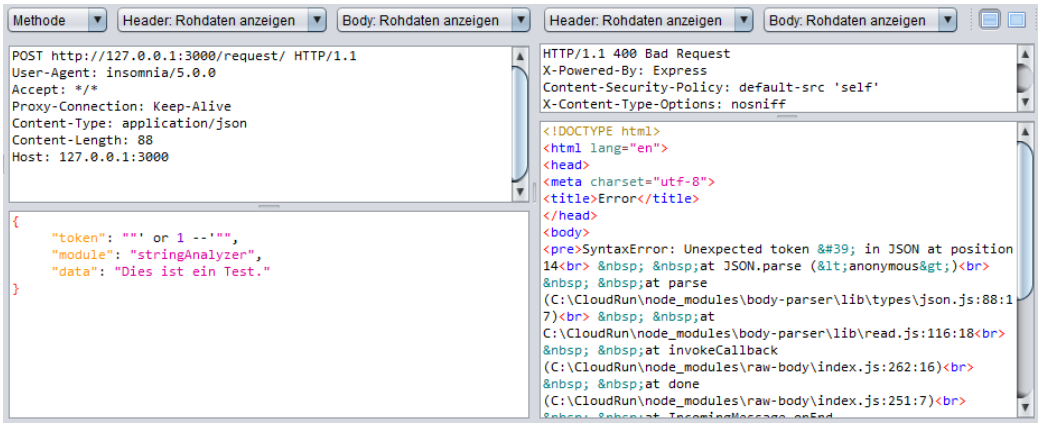
```
{
    "token": "test",
    "module": "stringAnalyzer",
    "data": "Dies ist ein Test!"
}
```

Daher werde ich das Fuzzen am `token`- sowie `module`-Parameter anwenden. Würde man das Fuzzing an beiden Parametern gleichzeitig anwenden, würde die Anzahl der zu versendenden Anfragen exponentiell ansteigen, weshalb jeder Parameter einzeln getestet werden sollte. Kreuzreaktionen sind sehr unwahrscheinlich und können daher vernachlässigt werden. Zunächst wird der `token`-Parameter untersucht.

Der Fuzzer von OWASP ZAP unterstützt keine automatische Erkennung von Sicherheitsrisiken, weshalb dieser Schritt manuell durchgeführt werden muss. Dazu betrachtet man die Liste der Anfragen und sucht (im Vergleich zu einer „echten“ Anfrage) nach Auffälligkeiten. Da bei diesem Durchlauf über 1500 Anfragen verschickt wurden, ist dabei eine Selektion nötig.

Die wichtigsten Anhaltspunkte sind dabei der HTTP-Statuscode sowie die Größe der Antwort in Bytes. Eine korrekte Antwort auf die ursprüngliche Anfrage wäre 84 Byte groß, jedoch enthält die Liste sehr viele Antworten mit einer Größe von 978 Byte.

Abbildung 2.17: OWASP ZAP - Fuzzing - Stacktrace in Antwort



Betrachtet man diese Antworten genau (Abbildung 2.17), fällt sofort auf, dass die Antwort des CloudRun Servers einen vollständigen Stacktrace enthält. Ausgelöst wurde dies durch eine fehlerhafte JSON-Formatierung:

```
SyntaxError: Unexpected token " in JSON at position 14
    at JSON.parse (<anonymous>)
```

Selbstverständlich sollte ein Stacktrace niemals an den Benutzer ausgeliefert werden, da dieser sensible Informationen enthält. Nach einer kurzen Recherche, warum `express` (der von CloudRun verwendete HTTP-Server) im Fehlerfall mit einem Stacktrace antwortet, wird klar, dass NodeJS standardmäßig im Entwicklungsmodus läuft – es muss explizit bekannt gemacht werden, dass es sich bei dem System um ein produktives Umfeld handelt [StackOverflow].

Setzt man die Umgebungsvariable `NODE_ENV` auf „`production`“ und führt die zum Stacktrace führende Anfrage erneut auf, so wird kein Stacktrace mehr ausgegeben. Es sollte also unbedingt beachtet werden, dass NodeJS zuvor auf die produktive Nutzung umgestellt werden muss.

Um dies sicherzustellen, habe ich CloudRun um eine Warnungsausgabe ergänzt, sollte sich das System nicht im produktiven Modus befinden (Abbildung 2.18).

Neben den Antworten, welche einen Stacktrace enthalten, tauchen ansonsten nur Antworten mit einer Länge von 71 Byte auf. Diese Antworten enthalten lediglich die Nachricht, dass die Authentifizierung fehlgeschlagen ist – das erwünschte Verhalten bei unzulässigen Authentifizierungstokens.

Da keine weiteren Auffälligkeiten gefunden wurden, kann nun der `module`-Parameter gefuzzed werden. Dabei verwende ich die gleichen Fuzzing-Listen wie im Test zuvor.

Zwar werden bei fehlerhafter JSON-Formatierung weiterhin Fehler ausgelöst, jedoch wird nun kein Stacktrace mehr ausgegeben (Abbildung 2.19). Auch beim Fuzzing des `module`-Parameters konnten ansonsten keine weiteren Auffälligkeiten gefunden werden.

Mit Hilfe des Fuzzings konnte ebenfalls ein Sicherheitsrisiko aufgedeckt werden, in diesem Fall sogar ein schwerwiegendes Risiko. Da es sich dabei jedoch um ein vom CloudRun Betreiber zu verantwortendes Risiko handelt, warnt CloudRun nun vor dem Einsatz der Software im nicht-produktiven Modus. Dadurch sollte die

Abbildung 2.18: CloudRun Warnung - Nicht-produktives Umfeld

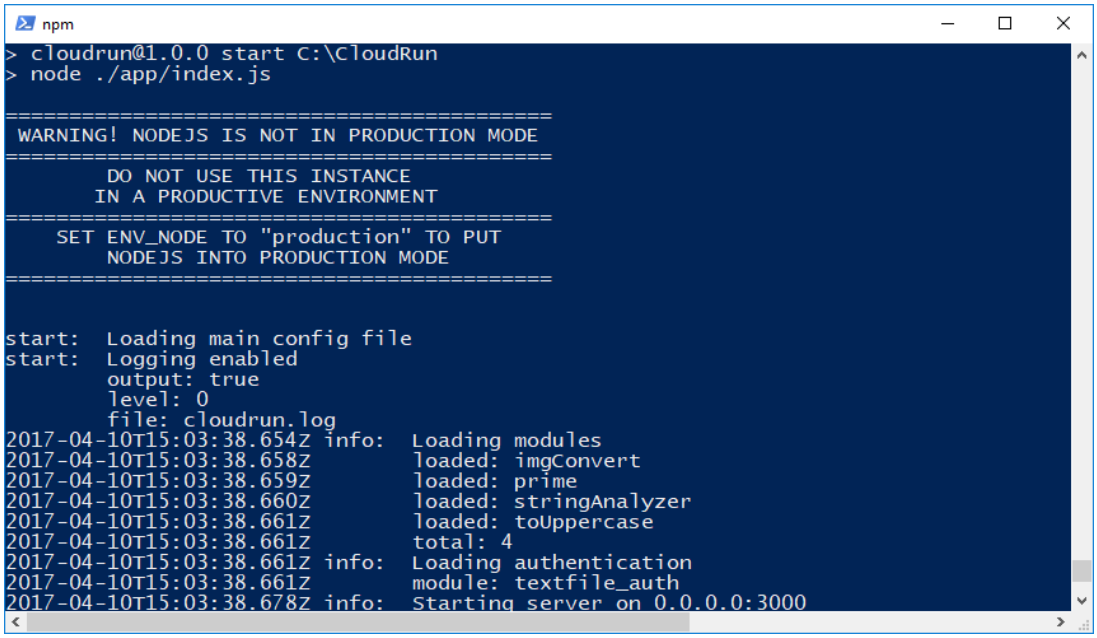
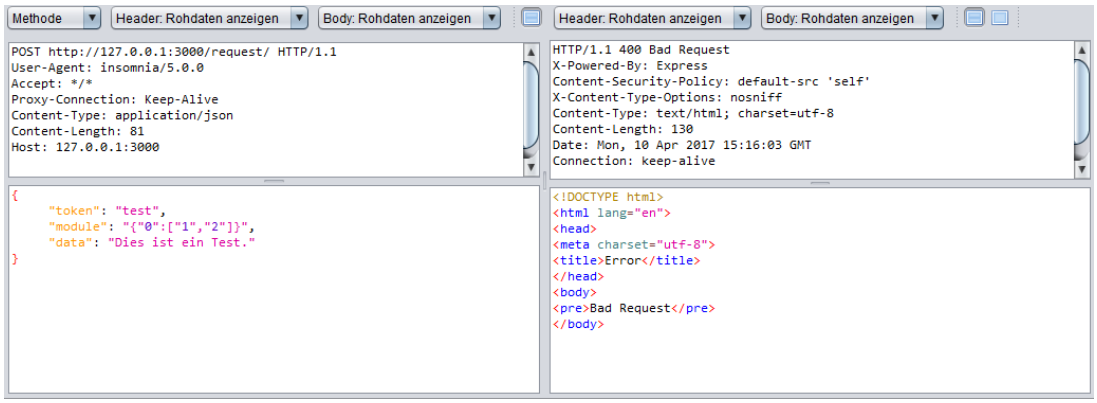


Abbildung 2.19: OWASP ZAP - Fuzzing - Kein Stacktrace in der Antwort



Wahrscheinlichkeit, dass dieses Problem im tatsächlichen Einsatz auftritt, deutlich minimiert worden sein.

Selbstverständlich könnte CloudRun auch die Ausführung verweigern, sollte sich das System nicht im produktiven Modus befinden, jedoch müsste man dieses Verhalten zu Entwicklungszwecken wieder entfernen. Es macht in meinen Augen mehr Sinn, die Verantwortung auf den Betreiber zu übertragen, da dieser nun sehr auffällig darauf

hingewiesen wird und für die Sicherheit des gesamten Systems verantwortlich ist.

2.2.5 Überprüfung des `mysql-auth` Authentifizierungsmoduls

CloudRun bietet die Möglichkeit, eine Tokenauthentifizierung mit Hilfe einer MySQL-Datenbank durchzuführen. Grundsätzlich bringt die Verwendung von Datenbanksystemen immer das Risiko von SQL-Injections mit sich, weshalb dieses Authentifizierungsmodul gesondert überprüft werden sollte.

SQL-Injections sind Angriffspunkte, an denen sich die ursprünglichen SQL-Befehle (welche an den Server versendet werden) manipulieren lassen. Dies kann sehr schwerwiegende Folgen haben, da sich – je nach Schwere der Sicherheitslücke – komplette Datenbankabbilder anfertigen lassen können. Außerdem ermöglichen SQL-Injections in den meisten Fällen die Möglichkeit der unberechtigten Authentifizierung.

Für die Überprüfung der MySQL-Authentifizierung verwende ich die Software „sqlmap“. Diese prüft automatisch, ob eine Webanwendung anfällig für SQL-Injections ist und bietet – sollte ein Einfallstor gefunden worden sein – weitere, sehr mächtige Funktionen zur unberechtigten Analyse der Datenbank.

Zur Vorbereitung wird die Datei `req.txt` angelegt. Diese Datei enthält einen HTTP-Request, bestehend aus HTTP-Headern und den Daten.

```
POST http://127.0.0.1:3000/request/ HTTP/1.1
Content-Type: application/json
Accept: */*
User-Agent: insomnia/4.2.14
content-length: 94
Connection: keep-alive
Host: 127.0.0.1:3000

{
  "token": "falscherToken",
  "module": "stringAnalyzer",
  "data": "Dies ist ein Test!"
}
```

Durch die Angaben in dieser Datei weiß sqlmap, welche Daten an welchen Server versendet werden sollen. Der darin angegebene Token ist bewusst falsch, da sqlmap standardmäßig nach Server-Antworten sucht, die von der Antwort der definierten

Anfrage abweichen (im Fall von CloudRun „Authentication failed“-Fehlernachrichten).

Ruft man sqlmap nun mit den geeigneten Parametern auf, so fängt die Software an nach möglichen SQL-Injections zu suchen.

```
python sqlmap.py -r req.txt -p token --level 5 --risk 3 --dbms mysql
```

Der Parameter `-r req.txt` gibt an, dass die zuvor erstellte Datei als Vorlage für die Anfragen verwendet werden soll. Dadurch ist auch direkt bekannt, welche URL angegriffen wird. `-p token` teilt sqlmap mit, dass der JSON-Parameter `token` manipuliert werden soll, um eine SQL-Injection aufzudecken. Der Parameter `--level 5 --risk 3` sorgt dafür, dass sqlmap dabei so aggressiv wie möglich vorgeht. Da bekannt ist, dass MySQL als Datenbanksystem verwendet wird, wird dies über `--dbms mysql` bekannt gemacht – dies dient lediglich der Minimierung des Testumfangs.

Nun beginnt sqlmap und gibt nach kurzer Zeit das Ergebnis aus. In diesem Falle ist das Authentifizierungsmodul `mysql_auth` tatsächlich angreifbar, was durch folgende Ausgabe mitgeteilt wird:

```
sqlmap identified the following injection point(s) with a total of 589
    HTTP(s) requests:
---
Parameter: JSON token ((custom) POST)
    Type: boolean-based blind
    Title: OR boolean-based blind - WHERE or HAVING clause
    Payload: {
        "token": "-6811" OR 1571=1571-- bolA",
        "module": "stringAnalyzer",
        "data": "Dies ist ein Test!"
}

    Type: AND/OR time-based blind
    Title: MySQL >= 5.0.12 OR time-based blind
    Payload: {
        "token": "falscherToken" OR SLEEP(5)-- RMVn",
        "module": "stringAnalyzer",
        "data": "Dies ist ein Test!"
}
---
[22:42:24] [INFO] the back-end DBMS is MySQL
back-end DBMS: MySQL >= 5.0.12
```

Dieses Sicherheitsrisiko ist als kritisch einzustufen. Um das Ergebnis zunächst zu überprüfen, sende ich die von sqlmap manipulierte Anfrage, welche zu einer Ausführung des Moduls führen sollte, manuell an den CloudRun Server. Dabei ist zu beachten, dass die von sqlmap ausgegebene Anfrage eine falsche JSON-Formatierung aufweist, welche in der folgenden Anfrage bereits korrigiert wurde.

```
{
        "token": "-6811\" OR 1571=1571-- bolA",
        "module": "stringAnalyzer",
        "data": "Dies ist ein Test!"
}
```

Obwohl kein gültiger Token angegeben wurde, erhält man das korrekte Ergebnis des Moduls `stringAnalyzer` als Antwort. Dies bestätigt, dass das Authentifizierungsmodul tatsächlich mittels einer SQL-Injection angreifbar ist.

Der als „falscher Token“ versandte String führt dazu, dass die an den MySQL-Server übertragene Anfrage manipuliert wird. Schaut man sich die entsprechende Code-Zeile des Authentifizierungsmoduls an, wird klar, wieso dies möglich ist.

```
connection.query('SELECT enabled FROM ' + prefix + 'token WHERE token =
    \"' + token + '\" AND enabled = 1 LIMIT 1', function (error, result)
     {
```

Die Variable `token` wird in diesem Fall direkt aus der JSON-Anfrage entnommen. Fügt man in die SQL-Anfrage anstatt der Variable den manipulierten String ein, so erkennt man die Funktionsweise der SQL-Injection:

```
SELECT enabled FROM token
WHERE token = \"-6811\" OR 1571=1571-- bolA\" AND enabled = 1 LIMIT 1
```

Der logische Ausdruck `token = [beliebig] OR 1=1` führt immer zu einer wahren Aussage, weshalb die Ausführung des Moduls mittels dieser SQL-Injection immer möglich ist. Um keine SQL-Fehler zu erzeugen, wird mit dem doppelten Bindestrich der restliche Teil der Anfrage auskommentiert.

Um dies zu verhindern, müssen alle vom Nutzer übermittelten und in die SQL-Anfrage eingebetteten Variablen „escaped“ werden [mysqljs Dokumentation]. Dazu

muss die oben erwähnte Codezeile folgendermaßen angepasst werden:

```
connection.query('SELECT enabled FROM '+prefix+'token WHERE token = '+
    connection.escape(token)+' AND enabled = 1 LIMIT 1', function (error
    , result) {
```

Durch den Aufruf der Funktion `connection.escape()` wird der übergebene Parameter automatisch für die SQL-Query vorbereitet, sodass keine SQL-Injections mehr möglich sind. Prüft man dies, indem man die zuvor funktionsfähige Anfrage erneut verschickt, stellt man fest, dass man nun korrekterweise eine „Authentication failed“-Nachricht erhält.

Um jedoch absolut sicher zu sein, dass keine SQL-Injections mehr möglich sind, habe ich sqlmap erneut ausgeführt. Auch hier kommt man zu dem Ergebnis, dass das kritische Sicherheitsrisiko behoben wurde.

Grundsätzlich entstehen Angriffspunkte für SQL-Injections durch eine nachlässige Programmierung, da dieses Sicherheitsrisiko sehr bekannt ist. Es genügt jedoch nur eine einzige, angreifbare SQL-Anfrage um möglicherweise vollen Zugriff zu erhalten, weshalb im Rahmen einer Sicherheitsanalyse stets auf SQL-Injections geprüft werden muss.

2.2.6 Sicherheitsrisiken innerhalb von Modulen

Da CloudRun auf seine Sicherheit überprüft wurde, kann nun auf mögliche Sicherheitsrisiken innerhalb von Modulen eingegangen werden. Dies ist ein besonders wichtiger Punkt, denn die Sicherheit eines CloudRun Servers kann nur gewährleistet werden, wenn auch die darauf installierten Module korrekt und sicher arbeiten. CloudRun hat keinen Einfluss auf die Arbeitsweise der Module, weshalb der Betreiber des Servers stets auf die Herkunft und Sicherheit der Module achten sollte.

Betrachten wir das Modul „`prime`“ aus der Zuverlässigkeitsanalyse (Kapitel 2.1.1) als Beispiel. Dieses Modul ruft eine ausführbare Datei auf und übergibt als Parameter die in der Anfrage definierten Werte.

```
const convert = exec(
  'call "C:\\CloudRun\\YACOBPrime.exe" ' // Aufruf der EXE
      +data.from.toString()              // von
```

```
      +' '+data.to.toString()            // bis
      +' '+data.threads.toString()+'',   // Anzahl Threads
  { encoding: "latin1" }, // Enkodierung in latin1
  function(err, stdout, stderr) {
```

Würde man nun einen manipulierten String für den Parameter `threads` übergeben, könnte man weitere Konsolenbefehle ausführen. Eine manipulierte Anfrage sähe beispielsweise so aus:

```
{
    "token": "",
    "module": "prime",
    "data": {
        "from": 1,
        "to": 100,
        "threads": "2 & echo ANGRIFF"
    }
}
```

Auf diese Anfrage erhält man die folgende Antwort:

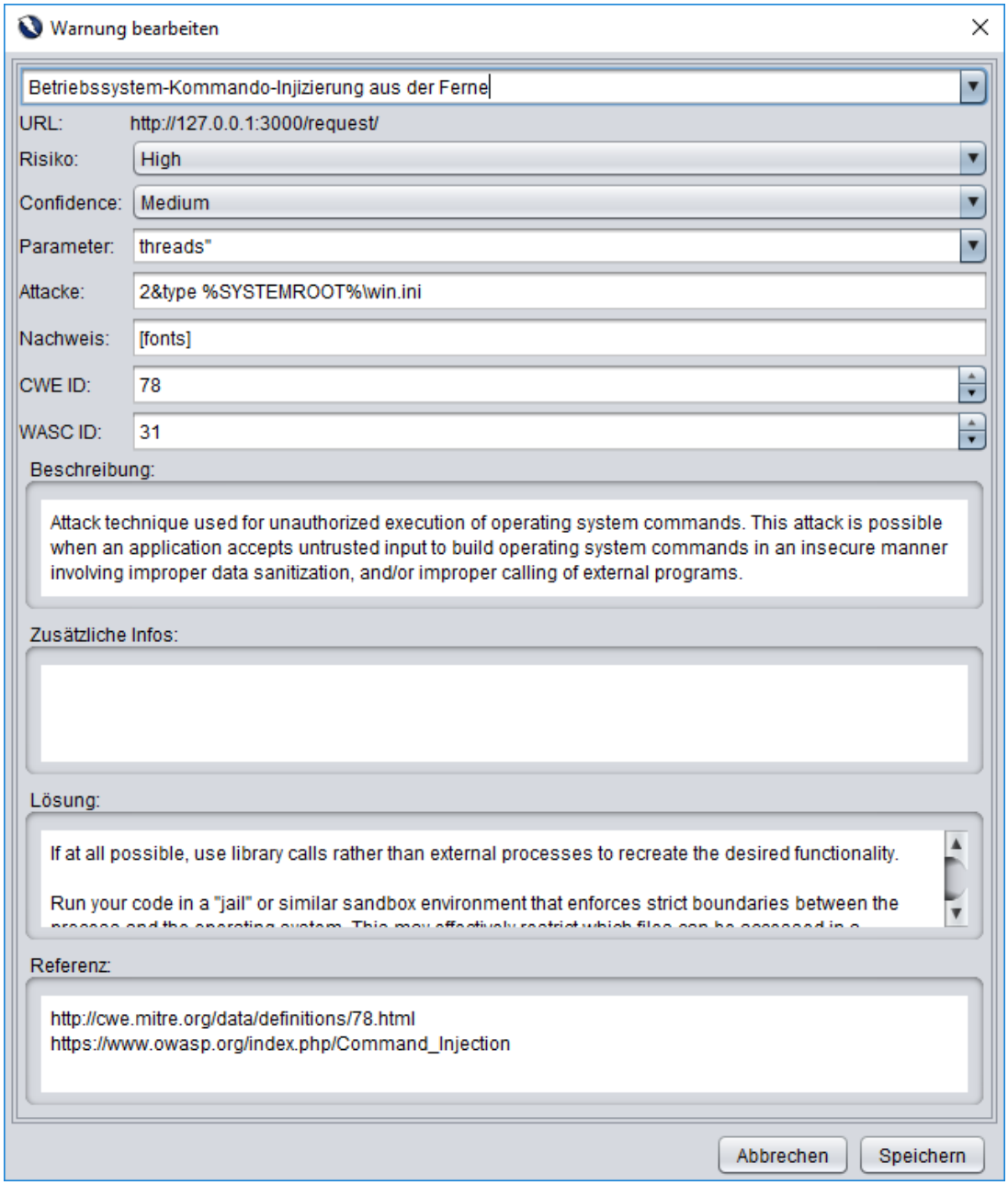
```
{
  "answer": "YACOB Prime started. Range: 1 - 100 with 2 Threads.\r\
    nAdding task 0: 1 - 100 with 2 steps\r\nAdding task 1: 2 - 100 with
    2 steps\r\nFinished.\r\nANGRIFF\r\n"
}
```

Hinter der Ausgabe von YACOBPrime taucht der Text „ANGRIFF“ auf, welcher durch die Ausführung des Befehls `echo Test` in der Anfrage erzeugt wird. Dies bedeutet, dass über das Modul `prime` beliebig viele Konsolenbefehle ausgeführt werden können, was selbstverständlich ein extremes Sicherheitsrisiko darstellt.

Um solche Sicherheitslücken zu vermeiden, darf grundsätzlich niemals ein Befehl ausgeführt werden, welcher ungeprüfte, vom Benutzer übertragene Parameter enthält. Würde man beispielsweise vor der Ausführung des Befehls prüfen, ob alle übergebenen Parameter von Typ „Integer“ sind, ließe sich ein solches Sicherheitsrisiko vermeiden.

Eine solche Sicherheitslücke lässt sich ebenfalls mittels eines aktiven Scans durch OWASP ZAP aufdecken (Abbildung 2.20), daher ist es grundsätzlich sinnvoll ein Modul vor der Veröffentlichung ebenfalls einer solchen Sicherheitsanalyse zu unterziehen.

Abbildung 2.20: OWASP ZAP - Sicherheitslücke im Modul `prime`



Zudem sollte der Betreiber eines CloudRun Servers stets darauf achten, ausschließlich Module aus bekannter und vertrauenswürdiger Herkunft zu installieren. Sollte nicht bekannt sein, ob das jeweilige Modul bereits einer Sicherheitsanalyse unterzogen wurde, ist die eigenständige Durchführung von Sicherheitstests ratsam, um die Sicherheit des Servers gewährleisten zu können.

2.2.7 Evaluierung der Ergebnisse

Die durchgeführte Sicherheitsanalyse konnte einige Schwachstellen CloudRuns, darunter ein kritisches Sicherheitsrisiko, aufdecken. Da diese Schwachstellen jedoch erfolgreich behoben werden konnten, führte die Analyse zu einer sicher einsetzbaren Software.

Da nun bekannt ist, dass CloudRun verschiedensten Sicherheitstests standhält, wäre ein produktiver Einsatz der Software grundsätzlich möglich. Wie das Zitat von Bruce Schneier in der Einleitung dieses Kapitels jedoch andeutet, kann niemals behauptet werden, dass eine Software zu 100% sicher ist, da in der Realität kein absolut sicheres Computersystem existiert.

Daher sollte stets beachtet werden, dass der Einsatz jeglicher Serveranwendungen jederzeit zu einer Beeinträchtigung der Sicherheit führen kann, dies gilt jedoch auch für den Einsatz bekannter Anwendungen (zum Beispiel „Apache“), in denen regelmäßig neue Sicherheitsrisiken entdeckt werden.

Der Quellcode CloudRuns kann öffentlich eingesehen werden. Zwar bietet Open Source Software im empirischen Vergleich keine erhöhte Sicherheit gegenüber Closed Source Software [Schryen, 2011], jedoch ermöglicht dies, dass ein Anwender mit dem nötigen Hintergrundwissen vor dem Einsatz CloudRuns prüfen kann, ob die Software Sicherheitsrisiken enhält, und kann darauf basierend entscheiden, ob CloudRun unter seiner Verantwortung zum Einsatz kommt. Zudem kann Jedermann zur Sicherheit CloudRuns beitragen, indem gefundene Sicherheitslücken an die Entwickler gemeldet werden.

Zusammenfassend lässt sich also sagen, dass CloudRun grundsätzlich ohne größere Bedenken produktiv eingesetzt werden kann. Dabei sollte jedoch stets darauf geachtet werden, dass die installierten Module ebenfalls sicher sind, da diese potenziell das größte Sicherheitsrisiko darstellen.

3 Einbindung in Python

Bei der Entwicklung von CloudRun stand stets im Fokus, dass dessen Benutzung so einfach wie möglich sein soll, um die Integration in neue und existierende Projekte nicht unnötig zu erschweren. Um dieses Ziel zu erreichen, werden Bibliotheken benötigt, die die Anbindung an einen CloudRun-Server übernehmen. Mit Hilfe einer solchen Bibliothek ließen sich ohne großen Aufwand Module, die auf dem CloudRun-Server liegen, in lokale Programme einbauen, ohne dass man sich selbst um den Verbindungsaufbau sowie die Anfragen- und Antwortaufbereitung kümmern muss.

Eine solche Bibliothek für die Programmiersprache Python wird in diesem Kapitel vorgestellt. Sie ermöglicht die Entwicklung von modulspezifischen Bibliotheken ohne den zusätzlichen Ballast, der durch die entfernte Ausführung von Algorithmen und Software entsteht.

Ich habe mich für die Entwicklung einer solchen Bibliothek für die Programmiersprache Python entschieden, da Python sehr verbreitet im wissenschaftlichen Umfeld eingesetzt wird. Grund für dessen Verbreitung sind beispielsweise die sehr mächtigen Bibliotheken NumPy [NumPy] und SciPy [SciPy]. Zudem erlaubt Python durch einfachen Syntax und flexible Datenstrukturen eine schnelle und effiziente Entwicklung, was ebenso zu Pythons Beliebtheit beiträgt [Koepke].

Im Folgenden wird zunächst erklärt, wie eine solche Bibliothek grundsätzlich aufgebaut sein sollte, um alle nötigen Funktionen bereitzustellen. Hierbei wird ebenso behandelt, wie eine darauf aufbauende, modulspezifische Bibliothek aussehen könnte, um die Verwendung eines Moduls auf einem CloudRun-Server so einfach wie möglich zu gestalten. Zuletzt wird detailliert auf die Entwicklung der Python-Bibliothek für CloudRun eingegangen.

3.1 Grundsätzlicher Aufbau und Ablauf

Dieses Kapitel beschreibt die Idee und den Aufbau von Bibliotheken für die Verwendung von CloudRun im Allgemeinen. Dabei muss zunächst darauf eingegangen werden, wie die einzelnen Bibliotheken in Verbindung stehen und welchen Zweck sie erfüllen.

Abbildung 3.1: Kommunikationsablauf einer Anfrage

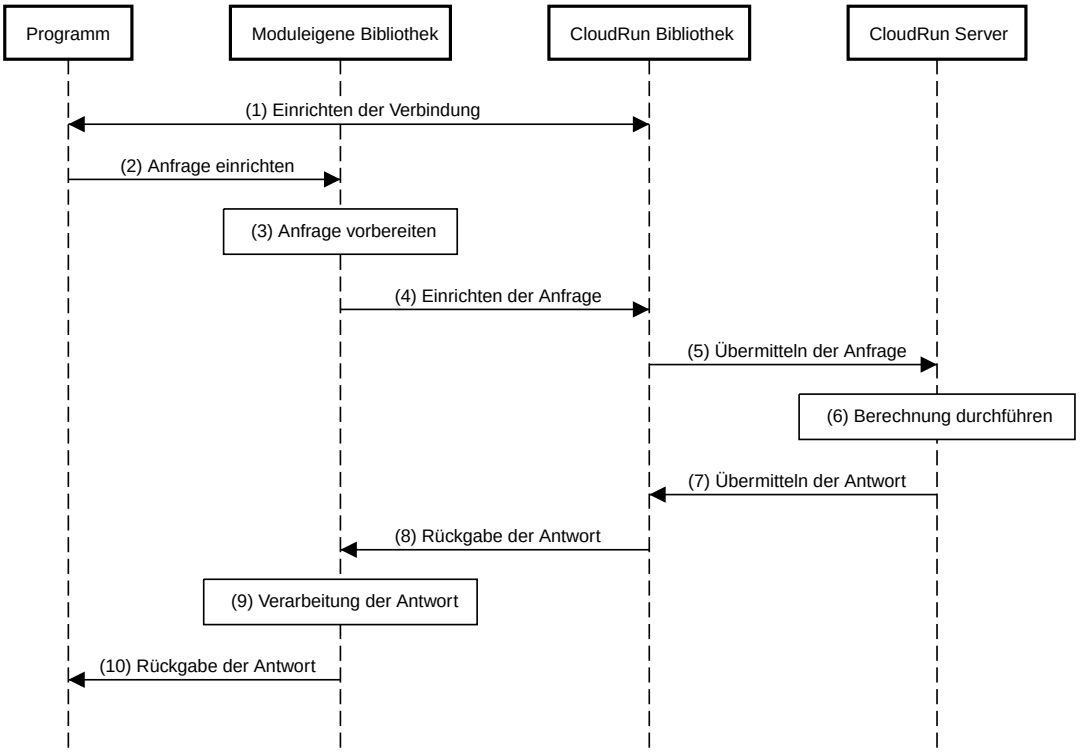


Abbildung 3.1 verdeutlicht die Kommunikation zwischen dem Programm, welches CloudRun verwendet, der modulspezifischen Bibliothek, der zu entwickelnden CloudRun-Bibliothek sowie dem CloudRun-Server. In diesem Beispiel gehen wir davon aus, dass das Programm eine Bilddatei bearbeiten soll, die Berechnung selbst soll jedoch auf dem CloudRun-Server erfolgen. Auf dem Server liegt ein dazu passendes Modul „ImageEdit“, welches Bilddateien hinsichtlich ihrer Helligkeit, Sättigung, etc. anpassen kann. Um weiter zu verdeutlichen, wie ein solcher Ablauf auf der Programmierebene aussehen kann, wird die Erläuterung von Pseudo-Codes begleitet.

Zunächst muss die Verbindung zum CloudRun-Server eingerichtet werden (1), dabei handelt es sich um die grundsätzlichen Informationen, also die URL, den Port und den Authentifizierungstoken.

```
// Verbindung zu http://cloudrun.de:80 mit Token "mein_token" herstellen
verbindung = new CloudRun("http://cloudrun.de", 80, "mein_token");
```

Im nächsten Schritt wird die auszuführende Anfrage definiert (2). Diese Aufgabe übernimmt die moduleigene Bibliothek. Um dem Beispiel zu folgen, wird also das zu bearbeitende Bild geladen, sowie die geplanten Veränderungen definiert.

```
// Anfrage erzeugen, 'verbindung' wird uebergeben
anfrage = new ImageEdit(verbindung);
anfrage.loadImage("[...]");          // Bild laden
anfrage.saettigung(0);               // Saettigung auf 0 setzen
anfrage.aufloesung(800, 600);        // Aufloesung auf 800x600 setzen
```

Während diese Befehle ausgeführt werden (3), erzeugt die modulspezifische Bibliothek im Hintergrund die Anfrage an den CloudRun-Server. Eine solche Anfrage könnte in diesem Falle folgendermaßen aussehen:

```
{
  'image': '[... Bild als BASE64 ...]',
  'saettigung': 0,
  'aufloesung': '800x600'
}
```

Diese Anfrage enthält jedoch keine Informationen über den Authentifizierungstoken sowie das aufzurufende Modul. Diese Informationen werden automatisch durch die CloudRun-Bibliothek ergänzt, welche in Schritt (1) bereits eingerichtet wurde.

```
// Anfrage abschicken
anfrage.send();
```

Die modulspezifische Bibliothek übergibt nun die vorbereitete Anfrage an die CloudRun-Bibliothek (4, innerhalb der modulspezifischen Bibliothek):

```
// Die Anfrage wird an das Modul "ImageEdit" gerichtet
request.module = "ImageEdit";

ergebnis = verbindung.sendPreparedRequest(request);
```

Es werden nun die folgenden Daten an den CloudRun-Server übermittelt (5):

```
{
  'token': 'mein_token',
  'module': 'ImageEdit',
  'data': {
    'image': '[... Bild als BASE64 ...]',
    'saettigung': 0,
    'aufloesung': '800x600'
  }
}
```

Der Server kann nun die Anfrage verarbeiten (6). Wurde die Berechnung fertiggestellt, so wird das Ergebnis zurück zur CloudRun-Bibliothek übertragen (7). Sollten keine Fehler aufgetreten sein, so wird das Ergebnis an die modulspezifische Bibliothek weitergeleitet (8), sodass sie verarbeitet werden kann (9). In dieser Verarbeitung wird beispielsweise die Umwandlung des Bildes aus dem Base64-Format in Binärdaten durchgeführt. Das Programm hat nun die Möglichkeit, die Antwort der verarbeiteten Anfrage auszulesen (10).

```
// Antwort auslesen
verarbeitetes_bild = anfrage.ergebnis();
```

Die Variable `verarbeitetes_bild` enthält nun das ursprüngliche Bild mit der Sättigung 0 in der Auflösung 800x600. Um zu verdeutlichen, wie wenig Aufwand für dieses Ergebnis betrieben werden muss, folgt nochmals der vollständige Pseudo-Code.

```
// Verbindung zu http://cloudrun.de:80 mit Token "mein_token" herstellen
verbindung = new CloudRun("http://cloudrun.de", 80, "mein_token");

// Anfrage erzeugen, 'verbindung' wird uebergeben
anfrage = new ImageEdit(verbindung);
anfrage.loadImage("[...]");          // Bild laden
anfrage.saettigung(0);               // Saettigung auf 0 setzen
anfrage.aufloesung(800, 600);        // Aufloesung auf 800x600 setzen

// Anfrage abschicken
```

```
11 anfrage.send();
12
13 // Antwort auslesen
14 verarbeitetes_bild = anfrage.ergebnis();
```

3.2 Theoretischer Aufbau der Bibliothek

Vor der Programmierung der Bibliothek ist es sinnvoll, eine theoretische Planung durchzuführen. Dies beinhaltet zum Einen eine Anforderungsanalyse, andererseits jedoch die konkrete Planung der Bibliothek, beispielsweise mittels UML-Diagrammen. Dieses Kapitel befasst sich damit, wie ich die Bibliothek im Voraus geplant habe, um später eine saubere und vollständige Implementation erreichen zu können.

3.2.1 Anforderungsanalyse und Planung

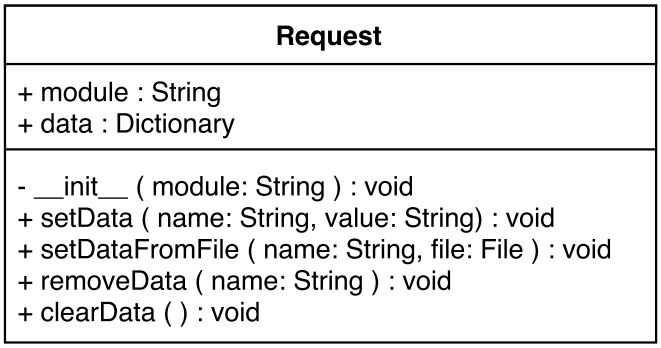
Eine Anforderungsanalyse ist nötig, um genau festlegen zu können, welche Funktionen die Bibliothek beinhalten muss. Dabei kann man sich an Anwendungsbeispielen (wie in Kapitel 3.1) und dem Funktionsumfang der anzubindenden Software, in diesem Falle CloudRun, orientieren.

Anfragen Die zu entwickelnde Bibliothek soll die Kommunikation zwischen einer lokalen Software und dem entfernten CloudRun-Server abstrahieren und technisch übernehmen. Da die einzelnen Module eines CloudRun-Servers jeweils selbst definieren, in welchem Format sie Daten entgegen nehmen möchten, ist es nötig die Anfragen an den Server individuell gestalten zu können, jedoch sollten Syntaxverletzungen (beispielsweise durch fehlerhaftes JSON) ausgeschlossen werden.

Dies bedeutet, dass die Bibliothek Funktionen bereitstellen muss, die die Erzeugung einer Anfrage ermöglichen. Sinnvoll wäre es daher, wenn sich Anfragen als Objekte darstellen lassen können. Eine dafür nötige Klasse, nennen wir sie `Request`, ließe sich als Container für die einzelnen Parameter der Anfrage darstellen.

Die Klasse `Request` (Abbildung 3.2) beinhaltet die Information, welches Modul auf dem CloudRun-Server angesprochen werden soll, sowie ein Dictionary (vgl. Array) `data`, welches die einzelnen Parameter der Anfrage beinhaltet. Die Methode `setData()` ermöglicht das Befüllen der Anfrage mit Parametern. Da CloudRun binäre Dateien Base64-kodiert überträgt, ist es sinnvoll eine Methode `setDataFromFile()`

Abbildung 3.2: Klasse `Request`

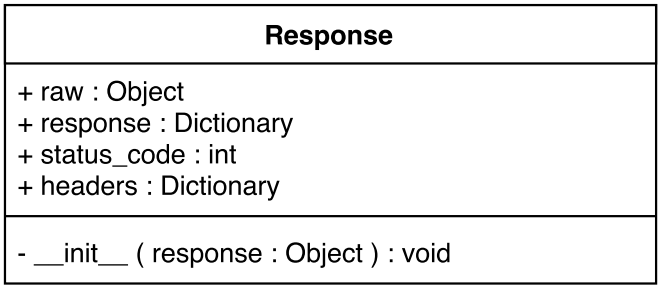


bereitzustellen, die eine Datei automatisch kodiert und zur Anfrage hinzufügt. Sollte der Fall eintreten, dass ein bereits gesetzter Parameter gelöscht werden soll, so steht die Methode `removeData()` bereit. Zusätzlich kann die Methode `clearData()` verwendet werden um den gesamten Inhalt der Anfrage zu löschen.

Antwort Da nun eine abstrakte Darstellung einer Anfrage definiert ist, liegt es nahe, auch eine Antwort abstrakt darzustellen. Grund dafür ist, dass ein direkter Umgang mit der Antwort des CloudRun-Servers fehleranfällig ist, da beispielsweise aufgetretene Fehler übersehen werden könnten. Diese Klasse nennt sich sinnvollerweise `Response`.

Da CloudRun jedoch nicht vorgibt, wie eine Antwort aussehen muss (außer das Datenformat JSON), um den Modulen die größtmögliche Freiheit zu lassen, muss eine Antwort ebenso allgemein behandelt werden. Jedoch werden durch das HTTP-Protokoll weitere Informationen übermittelt, die gegebenenfalls nützlich sein könnten, weshalb diese ebenso in der Antwort enthalten sein sollten.

Abbildung 3.3: Klasse `Response`



Die Klasse `Response` (Abbildung 3.3) beinhaltet vier Attribute. `raw` enhält die direkte Rückgabe der HTTP-Verbindung, welche jedoch nur verwendet werden sollte, wenn es nicht anders möglich ist. Das Dictionary `response` enhält die Rückgabe des CloudRun-Servers bzw. die Antwort des aufgerufenen Moduls. Die Attribute `status_code` und `headers` enthalten Informationen über die HTTP-Übertragung, welche eventuell von Bedarf sein können.

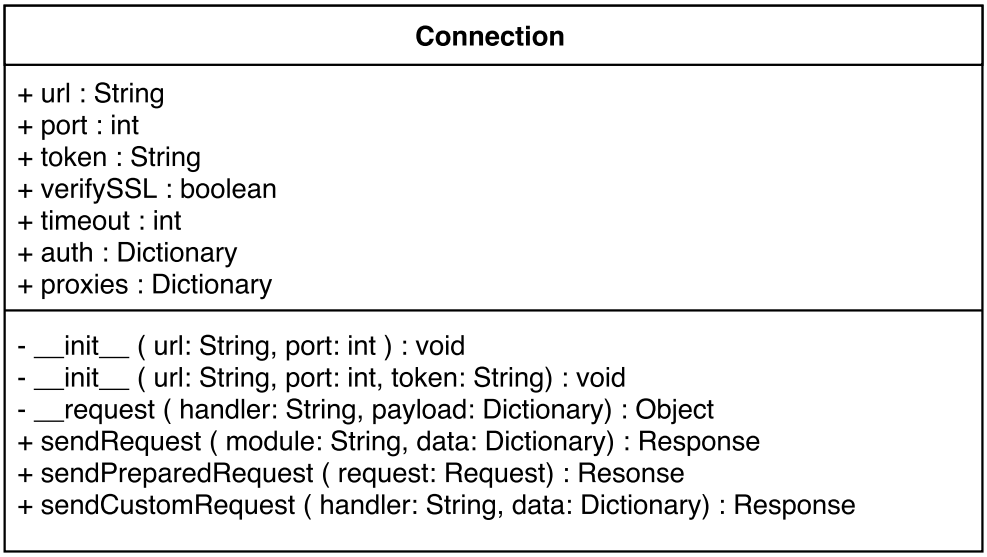
Verbindung Da nun die Containerklassen für Anfrage und Antwort bekannt sind, fehlt nun noch die eigentliche Logik der Bibliothek. Da in den allermeisten Fällen alle Anfragen eines Programms an den selben CloudRun-Server geschickt werden, macht es also Sinn die Logik in die Klasse zu implementieren, die sich auch um die Verbindung zum CloudRun-Server kümmert.

Zuerst muss festgestellt werden, welche Informationen nötig sind, um eine Verbindung zu einem CloudRun-Server herstellen zu können. Grundsätzlich ist dafür lediglich die Serveradresse sowie dessen Port nötig, weshalb für diese beiden Informationen ein Konstruktor existieren sollte. In vielen Fällen wird jedoch auch ein Authentifizierungstoken genutzt, daher ist es naheliegend einen zweiten Konstruktor anzubieten, der zusätzlich auch diese Information entgegen nimmt.

Die eigentliche Logik der Bibliothek wird mittels Methoden dieser Klasse bereitgestellt. Um den größtmöglichen Komfort, jedoch auch eine universelle Funktionalität zu gewährleisten, benötigt man am besten mehrere Methoden zum Verschicken einer Anfrage. Zum einen sollte man Anfragen selbst verfassen können, wenn möglicherweise die Nutzung der `Request`-Klasse (3.2) nicht möglich ist. Zum anderen sollte man jedoch auch die verschiedenen Handler aufrufen können, beispielsweise wenn Informationen über den CloudRun-Server abgerufen werden sollen (über den Handler „server_info“).

Neben den bereits erwähnten Attributen stellt die Klasse `Connection` (Abbildung 3.4) weitere Attribute bereit, die eventuell für den Verbindungsaufbau möglich sind. Das Attribut `verifySSL` bestimmt, ob die Zertifikate einer SSL-Verbindung auf ihre Gültigkeit überprüft werden sollen. Dies ist sinnvoll, wenn der CloudRun-Server zwar eine Verschlüsselung anbietet, die verwendeten Zertifikate jedoch nicht von einer vertrauenswürdigen Stelle signiert wurden. Das Attribut `timeout` definiert, nach wie vielen Sekunden die Anfrage abgebrochen werden soll, sollte keine Antwort vom

Abbildung 3.4: Klasse `Connection`



Server erfolgen.

`auth` enthält Informationen über die HTTP-Authentifizierung. Obwohl CloudRun selbst diese Option nicht bereitstellt, sollte die Möglichkeit in der Bibliothek bestehen, um universell einsetzbar zu sein. Zuletzt beinhaltet das Attribut `proxies` Informationen über die Verwendung eines Proxies für die HTTP-Verbindung. Diese beiden Attribute werden in Kapitel 3.3 näher erläutert.

Wie bereits erwähnt gibt es drei Methoden, die für das Abschicken von Anfragen zuständig sind. Die Methode `sendRequest()` ist für allgemeine Anfragen einsetzbar, bei denen kein `Request`-Objekt nötig bzw. möglich ist. Der Inhalt des `data`-Parameters ist daher frei definierbar. `sendPreparedRequest()` nimmt ein `Request`-Objekt entgegen und verschickt dieses an den Server. Zuletzt wird `sendCustomRequest()` bereitgestellt, welches für die Kommunikation mit anderen Handlern benutzt werden kann.

3.2.2 Abstrakte Darstellung der Bibliothek

Da nun die drei nötigen Klassen der Bibliothek festgelegt sind, werden diese nochmal in einem zusammenhängenden Klassendiagramm zusammengefasst (Abbildung 3.5). Zusätzlich lässt sich der Ablauf einer Anfrage nun als Objektflussdiagramm darstellen, um die Zusammenhänge der einzelnen Klassen bzw. Objekte weiter zu verdeutlichen (Abbildung 3.6).

Abbildung 3.5: Klassendiagramm der Bibliothek

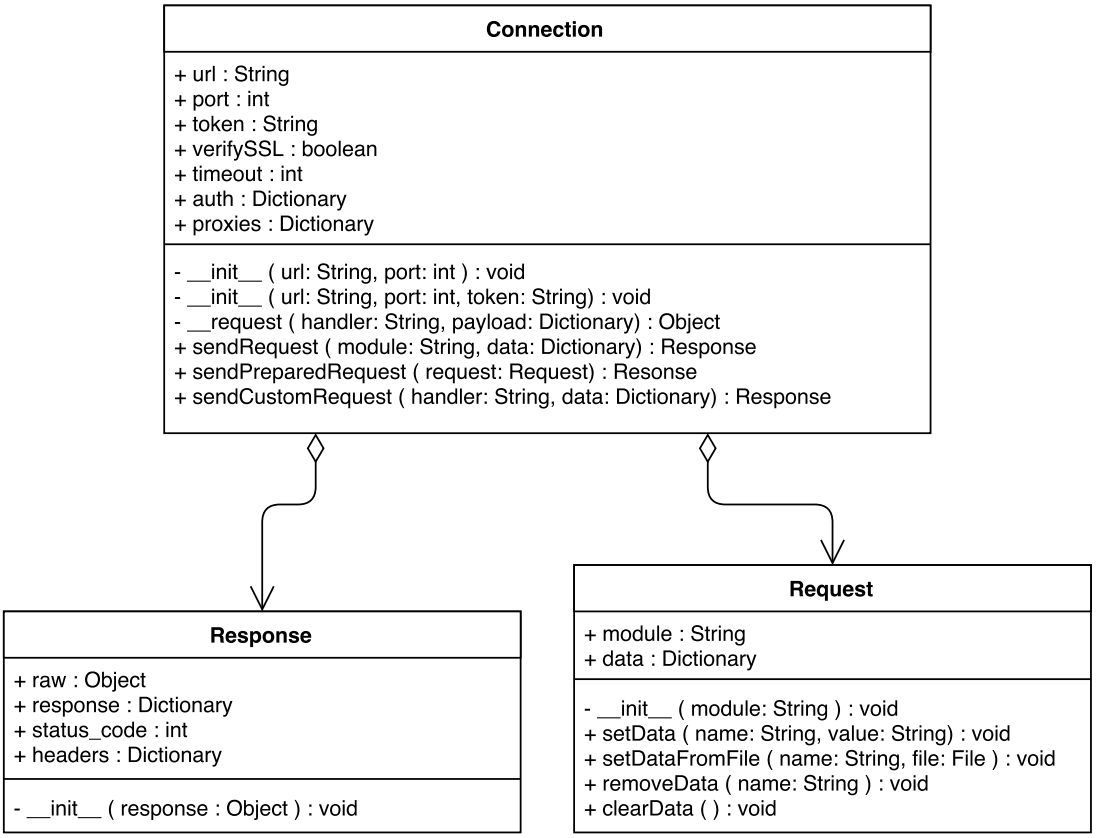
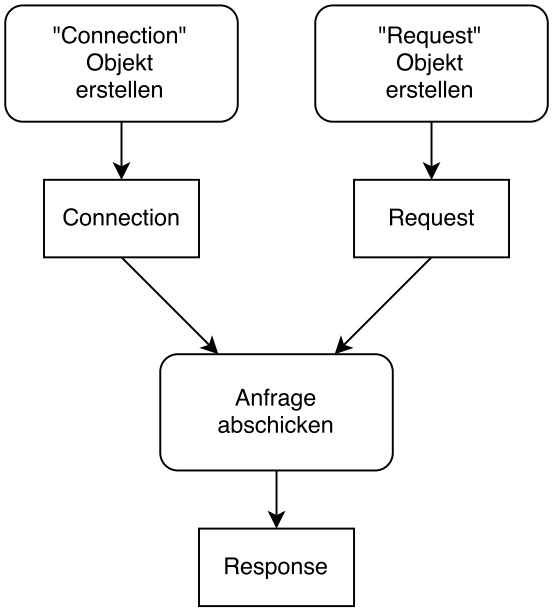


Abbildung 3.6: Objektflussdiagramm einer Anfrage



3.3 Implementation

Da der genaue Aufbau der zu entwickelnden Bibliothek vollständig geplant ist, kann nun mit der Implementation für die Programmiersprache Python begonnen werden. Im Folgenden wird zunächst auf die eigentliche Programmierung eingegangen und erläutert, was bei der Entwicklung einer solchen Bibliothek beachtet werden muss. Im Anschluss folgen Beispiele, wie die Bibliothek eingesetzt wird und welche Probleme dabei auftreten können.

3.3.1 Programmierung

Dieses Kapitel betrachtet detailliert die Entwicklung der bereits geplanten Klassen. Dabei werden aus Gründen der Übersichtlichkeit unwichtige Codestellen, beispielsweise Quelltext-Dokumentationen, ausgelassen. Der vollständige Quelltext der Bibliothek befindet sich im Anhang.

`Request`-Klasse Wie bereits bei der theoretischen Planung beginnen wir mit der Containerklasse für Anfragen an den CloudRun-Server.

```
1  class Request:
2      data = {}        # Empty data
3
4      def __init__(self, module):
5          self.module = module
6
7      def setData(self, name, value):
8          self.data[name] = value
9
10     def setDataFromFile(self, name, file):
11         # Datei auslesen und als BASE64 enkodieren
12         self.data[name] = base64.standard_b64encode(
13             file.read()).decode('ascii')
14         file.close()
15
16     def removeData(self, name):
17         del self.data[name]
18
19     def clearData(self):
20         self.data.clear()
```

Der Inhalt einer Anfrage wird in einem Dictionary `data` gespeichert. Ein Dictionary verhält sich wie ein assoziatives Array, vereinfacht beinhaltet es Schlüssel-Wert-

Paare. Da die Parameter, die an ein CloudRun-Modul gesendet werden ebenfalls als Schlüssel-Wert-Paare aufgebaut sind, ist ein Dictionary daher sehr gut geeignet. Zudem bietet es den Vorteil, dass es ohne großen Aufwand in einen JSON-String, der für die Übertragung an CloudRun nötig ist, übersetzt werden kann.

Der Konstruktor (Zeilen 4 bis 5) nimmt den Modulnamen als String entgegen, sodass beim Erstellen des Objekts bereits festgelegt werden kann, welches Modul angesprochen werden soll.

Die Methode `setData()` übernimmt das Einpflegen von Parametern in das Dictionary `data`. Prinzipiell ließe sich das Dictionary auch von außerhalb ansprechen, d.h. die Verwendung einer Methode wäre nicht nötig, jedoch fördert die Verwendung solcher Methoden die Lesbarkeit des Programmcodes erheblich.

Um das Übertragen von Dateien zu erleichtern, steht die Methode `setDataFromFile()` bereit. Als Argument übergibt man ihr den Parameternamen sowie ein File-Objekt. Die angegebene Datei wird automatisch ausgelesen und in einen Base64-ASCII-String konvertiert, sodass der Dateiinhalt an CloudRun übertragen werden kann. Um die Lesbarkeit des Programmcodes, welcher die CloudRun-Bibliothek verwendet, zu verbessern, wird ebenso das übergebene File-Objekt geschlossen, sodass dies nicht mehr im aufrufenden Programm geschehen muss.

Die Methode `removeData()` löscht einen Eintrag aus der Anfrage, `clearData()` hingegen löscht alle enthaltenen Einträge.

`Response`-Klasse Die Containerklasse `Response` ist sehr simpel aufgebaut, da sie lediglich die Aufgabe der Speicherung von erhaltenen Antworten übernimmt.

```
1  class Response:
2      raw = None           # Raw response if needed
3      response = None      # Dictionary of the response
4      status_code = None   # Status code
5      headers = None       # Response headers (dictionary)
6
7      def __init__(self, response):
8          self.raw = response
9          self.response = response.json()
10         self.status_code = response.status_code
11         self.headers = response.headers
```

Erhält man eine Antwort vom CloudRun-Server, so sind neben der eigentlichen Antwort des Moduls grundsätzlich keine weiteren Informationen von Interesse. Für den Fall einer Fehlfunktion ist es jedoch sinnvoll die Antwort des Servers genauer analysieren zu können.

Dazu enthält die `Response`-Klasse neben der Antwort des Moduls `response` als Dictionary auch den HTTP-Statuscode `status_code` sowie die übertragenen HTTP-Kopfzeilen `headers` als Dictionary. Zu Debuggingzwecken kann zusätzlich noch auf die „rohe“ Antwort der HTTP-Übertragung `raw` zugegriffen werden. Auf das Format der rohen Daten wird in der folgenden Erläuterung zur `Connection`-Klasse eingegangen.

`Connection`-Klasse Die eigentliche Logik der Bibliothek befindet sich in dieser Klasse. Sie übernimmt die Kommunikation mit dem CloudRun-Server und verarbeitet dessen Antworten. Sie stellt daher die Schnittstelle zwischen lokalem Programm und CloudRun-Server dar.

Python selbst stellt zwar Funktionen zum Herstellen einer HTTP-Verbindung bereit, jedoch fallen diese sehr rudimentär aus und sind daher ungeeignet für die Entwicklung einer Bibliothek für CloudRun. Für komplexere HTTP-Anfragen wird seitens der Python-Entwickler das Paket Requests (`http://python-requests.org`) empfohlen [Python Dokumentation, b], welches dank der Apache 2.0-Lizenzierung für alle Einsatzszenarien verwendet werden kann.

```
1  class Connection:
2      verifySSL = True     # Verify SSL-Cert?
3      timeout = None       # Timeout in seconds
4      auth = None          # HTTP-auth if necessary
5      proxies = None       # Proxies if necessary
6
7      def __init__(self, url, port, token=""):
8          self.url = url
9          self.port = port
10         self.token = token
```

Die Klasse `Connection` benötigt zwingend die `url` sowie den `port` des CloudRun-Servers. Neben diesen Informationen kann dem Konstruktor optional noch der Authentifizierungstoken `token` übergeben werden, sollte der Server eine Authentifizie-

rung voraussetzen.

Zudem werden vier weitere Optionen bereitgestellt, um die Verbindung zum CloudRun-Server genauer spezifizieren zu können. Weitere Informationen hierzu befinden sich in Kapitel 3.3.2.

```
1  def __request(self, handler, payload):
2      # POST-Request erzeugen und absenden
3      req = requests.post(
4              str(self.url) + ":" + str(self.port) + "/" +
5                  str(handler) + "/",
6              json=payload,
7              verify=self.verifySSL,
8              timeout=self.timeout,
9              auth=self.auth,
10             proxies=self.proxies)
11
12     req.raise_for_status() # Exception ausloesen wenn Statuscode Fehler
       anzeigt
13     answer = req.json(); # Antwort des Servers in Dictionary umwandeln
14
15     if 'error' in answer:   # If CloudRun answered with an error
16         raise CloudRunError("Error " + str(answer['error']['code']) + ":
        " + str(answer['error']['message']))
17         return None
18     else:
19         return req
```

Die private Methode `__request()` übernimmt die Kommunikation mit dem CloudRun-Server. Als Argumente nimmt sie den zu verwendenden Handler sowie die zu übertragenden Daten, die sogenannte `payload` entgegen.

Die Anfrage selbst wird über das bereits erwähnte Requests-Paket als POST-Request verschickt. Die aufzurufende URL der Anfrage ergibt sich aus der URL des CloudRun-Servers, dessen Port sowie dem zu verwendenden Handler (Zeilen 3 bis 4). Aus den Informationen

```
1 url = "http://cloudrun.de"
2 port = 3000
3 handler = "request"
```

ergibt sich dementsprechend die vollständige URL `http://cloudrun.de:3000/request/`.

Da CloudRun Daten im JSON-Format (genauer: `application/json`) entgegen nimmt, wird die Payload an den Parameter `json` übergeben (Zeile 5). Dadurch ist `requests` automatisch bekannt, in welchem Format die Daten übertragen werden sollen. Auch alle weiteren Optionen für die HTTP-Übertragung werden an dieser Stelle gesetzt (Zeilen 6 bis 9).

Sollte der zurückgegebene HTTP-Statuscode einen Fehler anzeigen [Network Working Group, 1999, Kapitel 10], so wird in Zeile 11 eine Exception erzeugt. Dies ermöglicht das Abfangen von auftretenden Fehlern mittels `try` und ermöglicht so eine saubere Verwendung der Bibliothek.

Da die Antwort des CloudRun-Servers ebenfalls im JSON-Format erfolgt, muss diese zur Weiterverarbeitung in ein Dictionary umgewandelt werden (Zeile 12). Sollte dieses Dictionary nun ein Feld „error“ enthalten, so hat der CloudRun-Server mit einer Fehlermeldung geantwortet. In diesem Fall wird eine `CloudRunError`-Exception ausgelöst, welche den Fehlercode sowie die Fehlernachricht enthält (Zeilen 14 bis 16). Eine `CloudRunError`-Exception ist eine benannte Exception [Python Dokumentation, a].

Sollte die Übertragung jedoch fehlerfrei abgelaufen sein, so wird das Antwortobjekt der HTTP-Anfrage zurückgegeben (Zeile 19).

```
1  def sendRequest(self, module, data):
2      payload = {'token': self.token, 'module': module, 'data': data}
3      req = self.__request("request", payload)
4
5      return Response(req)
6
7  def sendPreparedRequest(self, request):
8      if type(request) is Request:  # Wenn request vom Typ Request ist
9          return self.sendRequest(request.module, request.data)
10     else:
11         raise ValueError('request has to be a Request object')
12
13 def sendCustomRequest(self, handler, data={ }):
14     payload = {'token': self.token, 'data': data}
15     req = self.__request(handler, payload)
16     return Response(req)
```

Die Methode `sendRequest()` leitet eine Anfrage an den CloudRun-Server ein. Zu beachten ist jedoch, dass diese Methode kein `Request`-Objekt entgegen nimmt,

sondern den Modulnamen als String sowie die zu übertragenden Daten als Dictionary. Zunächst wird die Payload erzeugt, also der vollständige Inhalt der Anfrage. Dazu werden Token, Modulname und die Daten der Anfrage zu einem Dictionary zusammengefasst. Anschließend wird die Anfrage über die private Methode `__request()` an den Handler `request` des CloudRun-Servers verschickt. Die Antwort wird abschließend als `Response`-Objekt zurückgegeben.

Die darauf aufbauende Methode `sendPreparedRequest()` wird verwendet um eine vorbereitete Anfrage, also ein `Request`-Objekt an den CloudRun-Server zu schicken. Dabei wird zunächst geprüft, ob das übergebene Argument vom Typ `Request` ist, ansonsten wird eine `ValueError`-Exception ausgelöst. Sollte das übergebene Argument jedoch korrekt sein, so wird der Inhalt des `Request`-Objektes an die zuvor definierte Methode `sendRequest()` übergeben.

Möchte man jedoch statt des `request`-Handlers mit einem anderen Handler kommunizieren, so steht die Methode `sendCustomRequest()` zur Verfügung. Als Argumente nimmt sie den anzusprechenden Handler sowie die zu übertragenden Daten entgegen, welche jedoch optional sind. Wie bereits von der Methode `sendRequest()` bekannt wird zunächst die Payload gebildet, dann die private Methode `__request()` aufgerufen um abschließend dessen Ergebnis als `Response`-Objekt zurückzugeben.

Zusammenfassung Insgesamt war es möglich, die Implementation der CloudRun-Bibliothek sehr leichtgewichtig zu halten. Grund dafür ist das sehr einfach zu nutzende Paket Requests um HTTP-Anfragen zu verschicken, da Sie neben dem gesamten Verbindungsaufbau zusätzlich automatisch die Antwort in ein Python-typisches Format umwandelt. Dennoch beinhaltet die Bibliothek alle Funktionen die man für die Kommunikation mit einem CloudRun-Server benötigt und erfüllt daher alle zuvor festgelegten Anforderungen. Zudem lösen alle möglicherweise auftretenden Fehler (mindestens) eine Exception aus, was eine saubere und sichere Verwendung eines CloudRun-Servers aus einer Python-Anwendung heraus ermöglicht.

3.3.2 Anwendung

Da die Implementation der CloudRun-Bibliothek nun bekannt ist, folgen einige Anwendungsbeispiele. Dabei ist zu beachten, dass diese Beispiele teilweise imaginäre, also nicht existierende CloudRun Module verwenden, um den Fokus nicht auf die Module, sondern die Anwendung der Bibliothek zu richten.

Verwendung von vorbereiteten Anfragen Vorbereitete Anfragen, also Anfragen in Form eines `Request`-Objektes, sollten zu Gunsten einer sauberen Programmierung stets bevorzugt werden. Das folgende Beispiel bezieht sich auf das aus der Anforderungsanalyse bereits bekannte, imaginäre Modul „ImageEdit“. In diesem Fall soll die Verarbeitung eines Bildes nicht über eine gesonderte Bibliothek für dieses spezifische Modul erfolgen, sondern direkt über die nun bekannte CloudRun-Bibliothek.

Zu Beginn ist es sinnvoll, ein Verbindungsobjekt für den gewünschten CloudRun-Server zu erzeugen, jedoch muss die Bibliothek zunächst über einen `import` bekannt gemacht werden. Der CloudRun-Server läuft zu Testzwecken lokal unter dem Port 3000 und benötigt keine Authentifizierung. Das Erstellen eines Verbindungsobjektes kann keine Exception auslösen, weshalb keine Umklammerung mit `try` nötig ist.

```
1 import CloudRunPy
2
3 cloudrun = CloudRunPy.Connection("http://127.0.0.1", 3000)
```

Anschließend legen wir das `Request`-Objekt an und füllen es mit Daten.

```
1  request = CloudRunPy.Request("ImageEdit")
2
3  # Bild als "image" hinzufuegen, "rb" steht fuer binaeres Lesen
4  request.setDataFromFile("image", open("bild.jpg", "rb"))
5
6  # Rotation um 90 Grad im UZS
7  request.setData("rotate", 90)
8
9  # Helligkeit, Saettigung und Kontrast anpassen
10 request.setData("brightness", 120)
11 request.setData("saturation", 0)
12 request.setData("contrast", 130)
13
14 # EXIF-Daten sollen entfernt werden
15 request.setData("removeEXIF", true)
```

Möchte man die angelegte Anfrage nun auf dem CloudRun-Server ausführen lassen, so übergibt man diese der Methode `sendPreparedRequest()` des zuvor angelegten Verbindungsobjektes. Da nun eine tatsächliche Verbindung zum CloudRun-Server hergestellt wird, könnten im Falle eines Fehlers Exceptions ausgelöst werden, daher ist es nötig diese Anfrage mit einem `try`-Block zu umklammern.

```
try:
    answer = cloudrun.sendPreparedRequest(request)
```

Die Variable `answer` beinhaltet nun die Antwort des CloudRun-Servers als `Response`-Objekt. Da das verarbeitete Bild, welches vom Modul unter dem Namen „image“ zurückgegeben wird, derzeit jedoch Base64-enkodiert vorliegt, kann es nicht direkt weiter verarbeitet werden. In diesem Beispiel soll das Ergebnis als Datei abgespeichert werden und muss daher wieder ins binäre Format übersetzt werden. Dazu wird die Bibliothek „`base64`“ verwendet, welche zuvor importiert werden muss. Jedoch ist zu beachten, dass die Verarbeitung der Antwort nur möglich ist, wenn die Anfrage erfolgreich ausgeführt wurde, weshalb der folgende Code weiterhin im `try`-Block ausgeführt werden sollte. Zudem werden dadurch möglicherweise auftretende Fehler bei der Speicherung der Datei ebenfalls abgefangen.

```
    # Datei mit Schreibrechten binaer oeffnen
    f = open("ergebnis.jpg", "wb")
    f.write(base64.b64decode(answer.response['image']))
    f.close()
```

Sollten Exceptions ausgelöst worden sein, so müssen diese abgefangen werden. Das folgende Beispiel fängt explizit einen `CloudRunError` ab, andere Exceptions werden einfachheitshalber gesammelt verarbeitet.

```
except CloudRunError as err:
    print ("Es ist ein CloudRun-Fehler aufgetreten: " + str(err))
except:
    print ("Es ist ein unbekannter Fehler aufgetreten: " +
        str(sys.exc_info()[0]))
```

Für den Fall, dass beispielsweise das aufgerufene Modul auf dem CloudRun-Server nicht existiert, wird auf der Konsole folgende Nachricht ausgegeben:

```
Es ist ein CloudRun-Fehler aufgetreten: Error 3: Module not found
```

Betrachtet man nun den gesamten, zusammenhängenden Quelltext, so wird noch einmal deutlich, wie effizient die Benutzung der Bibliothek ist.

```
import CloudRunPy
import base64

cloudrun = CloudRunPy.Connection("http://127.0.0.1", 3000)

request = CloudRunPy.Request("ImageEdit")
request.setDataFromFile("image", open("bild.jpg", "rb"))
request.setData("rotate", 90)
request.setData("brightness", 120)
request.setData("saturation", 0)
request.setData("contrast", 130)
request.setData("removeEXIF", true)

try:
    answer = cloudrun.sendPreparedRequest(request)

    f = open("ergebnis.jpg", "wb")
    f.write(base64.b64decode(answer.response['image']))
    f.close()
except CloudRunError as err:
    print ("Es ist ein CloudRun-Fehler aufgetreten: " + str(err))
except:
    print ("Es ist ein unbekannter Fehler aufgetreten: " +
        str(sys.exc_info()[0]))
```

Verwendung von einfachen Anfragen Sollte die Verwendung von vorbereiteten Anfragen nicht möglich oder gewünscht sein, so lassen sich ebenfalls selbst konstruierte Anfragen verwenden. Dies hat grundsätzlich keine technischen Nachteile, ist jedoch unter Umständen schlechter lesbar und mindert dadurch auch die Wartbarkeit des Codes.

Ein mögliches Beispiel für eine Anfrage, die nicht über eine vorbereitete Anfrage ausführbar ist, stellt das bei CloudRun zu Testzwecken enthaltene Modul „stringAnalyzer“ dar. Dieses Modul erwartet als `data`-Attribut der Anfrage einen String, also kein Objekt welches die Daten enthält. Eine vorbereite Anfrage stellt jedoch immer ein Objekt dar, welches die Parameter enthält, und kann somit nicht verwendet werden.

Da bereits bekannt ist, wie die CloudRun-Bibliothek importiert und ein Verbindungsobjekt erstellt wird, werden diese Schritte übersprungen. Es wird angenommen, dass das `cloudrun`-Objekt bereits existiert. Zudem wird auf die Umklammerung mit `try` verzichtet. In diesem ersten Beispiel wird das zuvor erwähnte Modul „stringAnalyzer“ verwendet, um deutlich zu machen wie sich ein einzelner String an das Modul

verschicken lässt.

```
answer = cloudrun.sendRequest("stringAnalyzer",
         "Dieser String wird an das Modul uebertragen")
```

Verwendet man keine vorbereitete Anfrage, so kann die Anfrage dementsprechend direkt übertragen werden. Die an den CloudRun-Server übertragene Anfrage sieht in diesem Falle folgendermaßen aus:

```
{
    "token": "",
    "module": "stringAnalyzer",
    "data": "Dieser String wird an das Modul uebertragen"
}
```

Möchte man hingegen anstatt eines Strings ein Objekt, welches weitere Informationen enthält, übertragen, so wird ein Dictionary als Argument an `sendRequest()` übergeben. Dieses Beispiel verwendet das bereits bekannte, imaginäre Modul „ImageEdit“, dabei wird angenommen dass die Variable `b64image` das Bild bereits Base64-enkodiert enthält.

```
b64image = "[...]" # Base64-enkodiertes Bild
answer = cloudrun.sendRequest("stringAnalyzer",
         { 'image': b64image, 'rotate': 90 } )
```

Im Gegensatz zu vorbereiteten Anfragen ist diese Form wesentlich kompakter. Alternativ ließe sich die oben gezeigte Anfrage aber auch folgendermaßen darstellen, welche übersichtlicher, aber weniger kompakt ist:

```
b64image = "[...]" # Base64-enkodiertes Bild
request['image'] = b64image
request['rotate'] = 90
answer = cloudrun.sendRequest("stringAnalyzer", request)
```

Dabei fällt auf, dass diese Schreibweise der Verwendung von vorbereiteten Anfragen sehr nahe kommt und daher keine praktischen Vorteile bietet.

Ansprechen von alternativen Handlern CloudRun stellt neben Modulen, welche über den Handler „`request`“ ansprechbar sind, auch weitere Handler bereit. Stan-

dardmäßig enthält CloudRun beispielsweise den Handler „`server_info`“, welcher Informationen über den CloudRun-Server bereitstellt. Möchte man also beispielsweise überprüfen, ob ein bestimmtes Modul auf dem CloudRun-Server installiert ist, so ist eine Anfrage an diesen speziellen Handler nötig.

In dem folgenden Beispiel soll überprüft werden, ob der CloudRun-Server über das imaginäre Modul „ImageEdit“ verfügt. Wie im vorherigen Beispiel wurde die Bibliothek bereits importiert und das Verbindungsobjekt erstellt, ebenfalls wird auf eine Umklammerung mit `try` verzichtet.

```
answer = cloudrun.sendCustomRequest("server_info")
```

Über die Methode `sendCustomRequest()` wird der Handler `server_info` angesprochen. Zu beachten ist in diesem Falle, dass keine zu übermittelnden Daten als Argument übergeben werden, da dieses Argument optional ist und der Handler keine Daten benötigt.

Es wird in diesem Fall die folgende Anfrage an die URL `http://cloudrun.de:3000/server_info/` übertragen:

```
{
    "token": "",
    "data": ""
}
```

Die Antwort des Handlers `server_info` enthält unter anderem alle vorhandenen Module sowie deren Versionsnummern, weshalb auf folgende Weise die Existenz und – wenn vorhanden – die Version eines Moduls geprüft werden kann:

```
if ("ImageEdit" in answer.response['modules']):
    print("Vorhanden in Version " +
          str(answer.response['modules']['ImageEdit']))
else:
    print("Nicht vorhanden")
```

Verwendung von Proxies Da die CloudRun-Bibliothek auf das Paket Requests aufbaut, werden zur Herstellung der Verbindung zum CloudRun-Server auch Proxies unterstützt. Um die zu verwendenden Proxies bekannt zu machen, muss zunächst ein

Dictionary mit der URL bzw. IP sowie dem Port des Proxys für das jeweilige Protokoll (`http` oder `https`) angelegt werden [Requests Projektwebseite, Kapitel „Proxies“]. Anschließend wird das Dictionary an das Verbindungsobjekt der CloudRun-Bibliothek übergeben:

```
proxies = {
    'http':  'http://[user]@[password]:[ip]:[port]',
    'https': 'https://[user]@[password]:[ip]:[port]'
}

cloudrun.proxies = proxies
```

Weitere Informationen zur Verwendung von Proxies (u.a. auch die Verwendung von SOCKS-Proxies) sind in der Requests-Dokumentation abrufbar [Requests Projektwebseite, Kapitel „Proxies“].

Verwenden einer HTTP-Authentifizierung CloudRun stellt keine Möglichkeit einer HTTP-Authentifizierung bereit, jedoch ist es denkbar dass der CloudRun-Server über einen anderen Webserver weitergereicht wird (engl. „Relay“). Dabei besteht die Möglichkeit, eine HTTP-Authentifizierung einzurichten. Um eine solche Konfiguration mit Hilfe der CloudRun-Bibliothek zu Nutzen, müssen die Zugangsdaten zuvor an das Verbindungsobjekt übergeben werden.

Für eine HTTP-Basisauthentifizierung muss zunächst ein Objekt der Klasse `HTTPBasicAuth` des Requests-Paketes erstellt und anschließend an das Verbindungsobjekt der CloudRun-Bibliothek übergeben werden:

```
from requests.auth import HTTPBasicAuth
auth = HTTPBasicAuth('[user]', '[password]')

cloudrun.auth = auth
```

Das Requests-Paket bietet dabei für unterschiedliche HTTP-Authentifizierungsverfahren die nötige Unterstützung. Weitere Informationen zu diesem Thema sind in der Requests-Dokumentation abrufbar [Requests Projektwebseite, Kapitel „Authentication“].

Weitere Einstellungsmöglichkeiten CloudRun stellt noch zwei weitere Optionen zur Konfiguration der Verbindung bereit. Zum einen lässt sich ein Timeout einstellen, also die maximal erlaubte Dauer einer Anfrage in Sekunden. Wurde kein Wert explizit gesetzt, wird kein Timeout verwendet, was jedoch dazu führen kann, dass das Skript möglicherweise für mehrere Minuten hängt.

```
1 cloudrun.timeout = 60 # Timeout 60s
```

Weitere Informationen zu Timeouts sind in der Requests-Dokumentation abrufbar [Requests Projektwebseite, Kapitel „Timeouts“].

Standardmäßig unterstützt CloudRun keine verschlüsselten Verbindungen per HTTPS, jedoch wäre – ähnlich wie bei der HTTP-Authentifizierung – eine Verschlüsselung über ein Relay möglich. Bei der Verwendung eines CloudRun-Servers über eine HTTPS-Verbindung werden die Zertifikate der Verschlüsselung standardmäßig auf ihre Gültigkeit überprüft, jedoch lässt sich dies abschalten.

```
1 cloudrun.verifySSL = False # Ueberpruefung deaktivieren
```

Alternativ ist es auch möglich, vertrauenswürdige Zertifikate an das Requests-Paket zu übergeben um ausschließlich diese Zertifikate zuzulassen. Weitere Informationen zu diesem Thema sind in der Requests-Dokumentation abrufbar [Requests Projektwebseite, Kapitel „SSL Cert Verification“].

4 Fazit

Im Folgenden werden die Ergebnisse der einzelnen Untersuchungen und Entwicklungen im Rahmen dieser Bachelorarbeit zusammengefasst und resümiert. Anschließend folgt ein persönliches Fazit.

4.1 Fazit der Zuverlässigkeits- und Sicherheitsanalyse

Es stand die Frage im Raum, ob CloudRun in seiner ursprünglichen Form für den Einsatz in einer produktiven Umgebung geeignet ist – und wenn nicht, welche Änderungen getroffen werden müssen, um dies zu gewährleisten.

Im Rahmen des Kapitel 2.1, der Zuverlässigkeitsanalyse, wurde zunächst untersucht, wie belastbar ein CloudRun Server ist, insbesondere aber wie er auf starke Belastungen reagiert. Dabei konnte nicht nur herausgefunden werden, dass CloudRun sehr zuverlässig arbeitet, es konnte ebenso erfolgreich visualisiert und beschrieben werden, wie sich solch extreme Belastungen in den Antwortzeiten des Servers widerspiegeln.

Die Frage, ob ein CloudRun Server durch hohe Belastungen oder ggf. leichte Angriffe zu Fehlfunktionen oder sogar einem Absturz gebracht werden kann, kann klar verneint werden. Die Antwortzeiten des Servers verlängern sich den Umständen entsprechend teilweise zwar erheblich, jedoch ist dieses Verhalten durch die natürliche Begrenzung von Rechenkapazitäten nicht zu umgehen und daher für den realen Gebrauch CloudRuns nicht von Bedeutung.

Die von mir durchgeführte Zuverlässigkeitsanalyse bietet einen ersten Eindruck zur Belastbarkeit CloudRuns. Die mir persönlich zur Verfügung stehende Rechenleistung ist allerdings grundsätzlich nicht mit der Leistung eines vollwertigen Servers vergleichbar, weshalb eine erneute Durchführung einer solchen Zuverlässigkeitsanalyse mit leistungsstärkerer Hardware und noch höheren Belastungen eine sinnvolle Ergänzung darstellen würde.

In Kapitel 2.2, der Sicherheitsanalyse, wurde CloudRun anschließend mittels verschiedener Verfahren auf Sicherheitsrisiken untersucht. Dadurch, dass während dieser Analyse unter anderem eine kritische Sicherheitslücke aufgedeckt werden konnte, kann auch die Sicherheitsanalyse als Erfolg gewertet werden.

Zunächst wurde ein allgemeiner Sicherheitstest, der sogenannte „Active Scan“ der verwendeten Software „OWASP ZAP“, durchgeführt. Dabei konnte festgestellt werden, dass ein wichtiger HTTP-Header fehlt, was möglicherweise zu einer Angreifbarkeit hätte führen können. Dieser Fund stellt zwar keine wirkliche Sicherheitslücke dar, konnte allerdings auch sehr schnell behoben werden.

Durch die nächste Testmethode, dem Fuzzing, konnte ein als kritisch einzustufender Mangel aufgedeckt werden. Dabei wurde festgestellt, dass CloudRun bei fehlerhaft formatierten JSON-Anfragen mit einem vollständigen Stacktrace antwortet. Dieses Verhalten wurde nur indirekt durch CloudRun verursacht, da sich die NodeJS-Umgebung nicht im produktiven Modus befand und das verwendete express-Framework aus diesem Grund von einer Entwicklungsumgebung ausgeht. Um jedoch zu verhindern, dass dieses Problem auftritt, wenn CloudRun produktiv eingesetzt werden soll, wurde CloudRun dahingehend um eine auffällige Meldung beim Programmstart erweitert, sollte sich die NodeJS-Umgebung nicht im produktiven Modus befinden.

Eine kritische Sicherheitslücke konnte bei der Untersuchung des Authentifizierungsmoduls `mysql_auth` auf mögliche SQL-Injections entdeckt werden. Mit Hilfe der Software „sqlmap“ konnte ein möglicher Angriffspunkt für SQL-Injections festgestellt werden. Bei einer Ausnutzung dieser Sicherheitslücke hätte man den Server nicht nur ohne Berechtigung nutzen können, es wären auch deutlich schwerwiegendere Angriffe möglich gewesen. Auch dieses Risiko konnte erfolgreich behoben werden.

Mit Hilfe der durchgeführten Tests kann selbstverständlich nicht gewährleistet werden, dass CloudRun absolut sicher ist bzw. keine weiteren Sicherheitslücken existieren. Allerdings konnten mit Hilfe der Sicherheitsanalyse einige Probleme behoben werden, weshalb die potentielle Angreifbarkeit CloudRuns deutlich gesenkt werden konnte. Das Risiko der Verwendung CloudRuns muss jedoch weiterhin – wie bei jeder Serveranwendung – der Administrator des ausführenden Servers tragen.

Zusammenfassend betrachtet konnte mit Hilfe der beiden Analysen die Qualität CloudRuns nochmal deutlich gesteigert werden. Die Frage, ob CloudRun für den

produktiven Einsatz geeignet ist, kann nach den durchgeführten Anpassungen bejaht werden.

4.2 Fazit zur Entwicklung einer Anwendungsbibliothek

Aufgabe war, eine Bibliothek zu entwickeln, die es dem Benutzer eines CloudRun Servers ermöglicht, den bereitgestellten Dienst komfortabel nutzen zu können. Die Anforderungen an eine solche Bibliothek wurden zunächst in Kapitel 3.1 aufgestellt.

Kapitel 3.2 hat sich anschließend mit der theoretischen Planung einer solchen Bibliothek beschäftigt. Diese Planung ermöglichte mir nicht nur die Entwicklung einer Implementation für Python, sondern dient gleichzeitig auch anderen Entwicklern als Basis zukünftiger Implementationen für andere Programmiersprachen.

Die eigentliche Implementation für die Programmiersprache Python wurde dann in Kapitel 3.3 durchgeführt. Die vorhergehende Planung erwies sich dabei als vollständig und konnte ohne Auffälligkeiten oder Probleme umgesetzt werden. Die daraus resultierende Implementation bietet die Möglichkeit, auf alle Funktionen eines CloudRun Servers zuzugreifen und erfüllt daher die anfangs gestellten Anforderungen.

Auch die Entwicklung einer solchen Bibliothek hat zum Wert CloudRuns beigetragen, da CloudRun nun nicht mehr nur manuell mit Hilfe eines REST-Clients ansprechbar ist, sondern direkt in Programme integriert werden kann.

4.3 Persönliches Fazit und Ausblick

Ich bin mit den Ergebnissen meiner Bachelorarbeit sehr zufrieden. Die Entwicklung an CloudRun fand ich bereits zum Praxisprojekt sehr interessant, weil der potentielle Nutzen der Software meiner Meinung nach sehr groß ist. Auch NodeJS trägt sehr dazu bei, denn die Entwicklung damit ist sehr zielführend („straight forward“) und benutzerfreundlich, zudem ist NodeJS außerordentlich gut dokumentiert.

Die Zuverlässigkeits- und Sicherheitsanalyse war für mich ein Novum und erforderte viel Vorbereitung und Einarbeitung in das Thema. Dabei konnte ich nicht nur lernen, wie grundsätzlich bei solchen Analyse vorgegangen werden muss, auch die Kenntnisse über die Bedienung der verwendeten Software wird mir in Zukunft sehr von Nutzen

sein.

Ich habe gelernt, dass insbesondere die Sicherheitsanalyse unter Zuhilfenahme von geeigneter Spezialsoftware sehr wichtig ist, da sich viele Sicherheitsrisiken ohne tiefgründige Fachkenntnisse ansonsten nicht auffinden lassen. Dieses Wissen wird mir in Zukunft sehr helfen, um andere Projekte gegen potentielle Angriffe absichern zu können.

Während der Durchführung der Analysen traten keine nennenswerten Probleme auf. Die Bedienung der verwendeten Software war anfangs zwar etwas kompliziert, hat man jedoch das Grundprinzip erst einmal verstanden, so ist auch dies kein großes Problem mehr. Es fällt jedoch auf, dass solche Spezialsoftware in den meisten Fällen nur von dafür geschulten Experten eingesetzt wird – dementsprechend ist die Verbreitung von öffentlich zugänglichen, ausführlichen Anleitungen sehr dünn.

Auch die Entwicklung einer Bibliothek für CloudRun verlief weitgehend ohne Probleme, da ich über langjährige Programmiererfahrungen verfüge. Dabei hat sich auch bemerkbar gemacht, dass wir mit dem Konzept CloudRuns, also der Art und Weise wie damit kommuniziert wird, eine gute Arbeit geleistet haben. Dies mache ich daran fest, dass die Bibliothek trotz seiner vollständigen Unterstützung aller Funktionen CloudRuns relativ kompakt blieb. Es ist also nicht viel Aufwand nötig um mit einem CloudRun Server zu kommunizieren, was meiner Meinung nach ein guter Indikator für ein durchdachtes und funktionierendes Gesamtkonzept ist.

Für die Zukunft CloudRuns wünsche ich mir, dass das Projekt von anderen Studenten weiter entwickelt und gepflegt wird. Ich bin davon überzeugt, dass CloudRun gerade im akademischen Umfeld sehr von Nutzen ist und daher auch das Interesse weiterer Universitäten und Hochschulen wecken kann. Vorstellbar wäre daher eine Kollaboration verschiedener Institute zum Aufbau eines zusammenhängenden CloudRun-Netzwerkes. Diese Vorstellung ist gleichzeitig auch der Ausblick, denn ein solches Netzwerk ließe sich beim aktuellen Entwicklungsstand CloudRuns nur mit größerem Aufwand realisieren.

Wünschenswert wäre daher neben der Entwicklung eines dezentralen Authentifizierungssystems auch die Möglichkeit der automatischen Vernetzung von CloudRun Servern.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Algorithmia] Algorithmia: *Algorithmia Projektwebseite*. https://www.algorithmia.com. – Abgerufen am 21.04.2017

[Apache Software Foundation a] Apache Software Foundation: *Apache JMeter*. https://jmeter.apache.org/. – Abgerufen am 06.04.2017

[Apache Software Foundation b] Apache Software Foundation: *Apache JMeter User's Manual*. https://jmeter.apache.org/usermanual/. – Abgerufen am 06.04.2017

[Bundesamt für Sicherheit in der Informationstechnik] Bundesamt für Sicherheit in der Informationstechnik: *IT-Sicherheitskriterien und Evaluierung nach ITSEC*. https://www.bsi.bund.de/DE/Themen/ZertifizierungundAnerkennung/Produktzertifizierung/ZertifizierungnachCC/ITSicherheitskriterien/ITSEC/itsec_node.html. – Abgerufen am 02.04.2017

[Fielding 2000] Fielding, Roy T.: *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, Diss., 2000. https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. – Abgerufen am 23.02.2017

[github/fuzzdb-project] github/fuzzdb-project: *FuzzDB*. https://github.com/fuzzdb-project/fuzzdb. – Abgerufen am 10.04.2017

[Google] Google: *Google Chrome V8*. https://developers.google.com/v8/. – Abgerufen am 10.04.2017

[Internet Engineering Task Force (IETF) 2014] Internet Engineering Task Force (IETF): *RFC 7159 - The JavaScript Object Notation (JSON) Data Interchange Format*. https://www.w3schools.com/js/js_json_intro.asp. Version: March 2014. – Abgerufen am 23.02.2017

[Koepke] Koepke, Hoyt: *10 Reasons Python Rocks for Research (And a Few Reasons it Doesn't)*. https://www.stat.washington.edu/~hoytak/blog/whypython.html. – Abgerufen am 13.03.2017

[Kumar 2011] Kumar, Satish: *Decompilation.* `http://www.debugmode.com/dcompile/`. Version: Oct 2011. – Abgerufen am 21.04.2017

[mysqljs Dokumentation] mysqljs Dokumentation: *Escaping query values.* `https://github.com/mysqljs/mysql#escaping-query-values`. – Abgerufen am 11.04.2017

[Network Working Group 1999] Network Working Group: *RFC 2616, Hypertext Transfer Protocol – HTTP/1.1.* `https://tools.ietf.org/html/rfc2616`. Version: June 1999. – Abgerufen am 15.03.2017

[NodeJS] NodeJS: *NodeJS Projektwebseite.* `https://nodejs.org/`. – Abgerufen am 24.04.2017

[NumPy] *NumPy.* `http://www.numpy.org/`. – Abgerufen am 13.03.2017

[OWASP a] OWASP: *OWASP Attacks.* `https://www.owasp.org/index.php/Category:Attack`. – Abgerufen am 10.04.2017

[OWASP b] OWASP: *OWASP Zed Attack Proxy Project.* `https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project`. – Abgerufen am 10.04.2017

[Pahlen 2016] Pahlen, Andreas: *YACOB - Yet Another CPU Overclocking Benchmark.* `https://yacob.sourceforge.io`. Version: 2016. – Abgerufen am 05.04.2017

[Pahlen u. Förster 2017] Pahlen, Andreas ; Förster, Stefan: CloudRun - Cloudbasierte Ausführung von Algorithmen und Software / Technische Hochschule Köln. 2017. – Dokumentation Praxisprojekt

[Portswigger] Portswigger: *Burp Suite Editions.* `https://portswigger.net/burp/`. – Abgerufen am 10.04.2017

[Python Dokumentation a] Python Dokumentation: *Errors and Exceptions.* `https://docs.python.org/3/tutorial/errors.html`. – Abgerufen am 15.03.2017

[Python Dokumentation b] Python Dokumentation: *http.client — HTTP protocol client.* `https://docs.python.org/3/library/http.client.html`. – Abgerufen am 15.03.2017

[Requests Projektwebseite] Requests Projektwebseite: *Requests: HTTP for Humans.* `http://docs.python-requests.org`. – Abgerufen am 16.03.2017

[Schneier 1994] Schneier, Bruce: *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* Wiley John + Sons, 1994. – ISBN 0–471–59756–2

[Schryen 2011] Schryen, Guido: Is open source security a myth? What do vulnerability and patch data say? In: *Communications of the ACM (CACM)* Vol. 54 (2011), May, Nr. No. 5, 130–139. https://epub.uni-regensburg.de/21250/. – Abgerufen am 11.04.2017

[SciPy] *SciPy*. https://www.scipy.org/. – Abgerufen am 13.03.2017

[sqlmap Projektseite] sqlmap Projektseite: *sqlmap: automatic SQL injection and database takeover tool*. http://sqlmap.org/. – Abgerufen am 10.04.2017

[StackOverflow] StackOverflow: *Why does express's default error handler behaviour return stack traces to the client?* https://stackoverflow.com/questions/19541705/why-does-expresss-default-error-handler-behaviour-return-stack-traces-to-the-cl. – Abgerufen am 10.04.2017

[US-CERT] US-CERT: *Understanding Denial-of-Service Attacks*. https://www.us-cert.gov/ncas/tips/ST04-015. – Abgerufen am 10.04.2017

Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben.

Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Gummersbach, 10. Mai 2017

Andreas Pahlen

Anhang

Quellcode der Python-Bibliothek

```
import requests
import base64

class Connection:
    """
    Main class for the CloudRun-Connection
    """

    verifySSL = True    # Verify SSL-Cert?
    timeout = 60        # Timeout in seconds
    auth = None         # HTTP-auth if necessary
    proxies = None      # Proxies if necessary


    def __init__(self, url, port, token=""):
        """
        Construct a new 'Connection' object with an authentication token

        :param url: URL of the server, WITH "http://" or "https://"!
        :param port: Port of the server
        :param token: Your authentication token. If not set it defaults
    to ""
        :return: Returns nothing
        """

        self.url = url
        self.port = port
        self.token = token


    def __request(self, handler, payload):
        """
        Private method for a request

        :param handler: The handler that shall be used
        :param payload: What you want to send to the server
        :return: Returns the response from the request
        """
```

```
38
39          req = requests.post(str(self.url) + ":" + str(self.port) + "/" +
     str(handler) + "/",
40                              json=payload,
41                              verify=self.verifySSL,
42                              timeout=self.timeout,
43                              auth=self.auth,
44                              proxies=self.proxies)
45
46          req.raise_for_status()
47          answer = req.json();
48          if 'error' in answer:   # If CloudRun answered with an error
49              raise CloudRunError("Error " + str(answer['error']['code'])
    + ": " + str(answer['error']['message']))
50              return None
51          else:
52              return req
53
54
55      def sendRequest(self, module, data):
56          """
57          Send a request (to /request/ handler)
58
59          :param module: The module you want to use
60          :param data: The data you want to send to the module
61          :return: 'Response' object containing the response of the
    request
62          """
63
64          payload = {'token': self.token, 'module': module, 'data': data}
65          req = self.__request("request", payload)
66
67          return Response(req)
68
69
70      def sendPreparedRequest(self, request):
71          """
72          Send a prepared request (to /request/ handler)
73
74          :param request: 'Request' object contaning the request
    information
75          :return: 'Response' object containing the response of the
    request
76          """
77
78          if type(request) is Request:
79              return self.sendRequest(request.module, request.data)
```

```
        else:
            raise ValueError('request must be a CloudRunPy.Request')


    def sendCustomRequest(self, handler, data):
        """
        Send a custom request to a handler of your choice

        :param handler: The handler you want to use
        :param data: The data you want to send to the handler
        :return: 'Response' object containing the response of the
    request
        """

        payload = {'token': self.token, 'data': data}
        req = self.__request(handler, payload)
        return Response(req)



class Request:
    """
    Request class used for prepared requests
    """

    data = {}       # Empty data


    def __init__(self, module):
        """
        Construct a new 'Request' object

        :param module: The module you want to use
        :return: Returns nothing
        """

        self.module = module


    def setData(self, name, value):
        """
        Sets a data field to the specified value

        :param name: Name of the data field
        :param value: Value of the data field
        :return: Returns nothing
        """
```

```
126
127             self.data[name] = value
128
129
130         def setDataFromFile(self, name, file):
131             """
132             Sets a data field with the contents of a file (base64 encoded)
133
134             :param name: Name of the data field
135             :param file: the file you want to use
136             :type file: 'file' object (returned by open())
137             :return: Returns nothing
138             .. note:: Always open the file in binary mode!
139             """
140
141             self.data[name] = base64.standard_b64encode(
142                 file.read()).decode('ascii')
143             file.close()
144
145
146         def removeData(self, name):
147             """
148             Removes a data field
149
150             :param name: Name of the data field
151             :return: Returns nothing
152             """
153
154             del self.data[name]
155
156
157         def clearData(self):
158             """
159             Clears the data field
160
161             :return: Returns nothing
162             """
163
164             self.data.clear()
165
166
167
168 class Response:
169     """
170     Response class, contains the response
171     """
172
```

```
173     raw = None           # Raw response if needed
174     response = None      # Dictionary of the response
175     status_code = None   # Status code
176     headers = None       # Response headers (dictionary)
177 
178 
179     def __init__(self, response):
180         """
181         Construct a new 'Response' object
182 
183         :param response: The response of the request
184         :return: Returns nothing
185         """
186 
187         self.raw = response
188         self.response = response.json()
189         self.status_code = response.status_code
190         self.headers = response.headers
191 
192 
193 class CloudRunError(Exception):
194     """
195     CloudRunError Exception, will be raised when CloudRun submits an
    error
196     """
197 
198     pass
```